ESCUELA POLITECNICA NACIONAL

TRADUCTOR CROSS-ASSEMBLER PARA EL MICROPROCESADOR M6800

TESIS PREVIA A LA OBTENCION DEL TITULO DE INGENIERO EN LA ESPECIALIZACION DE ELECTRONICA Y TELECOMUNICACIONES.

MIRIAM B. HERNANDEZ ALVAREZ

QUITO-JULIO-1 9 8 2.

AGRADECIMIENTO

A mis profesores de la Escuela Politécnica Nacional y en forma particular, al Ing. Edgar P. Torres P. Director de mi Tesis, cuya colaboración y valiosa dirección, hizo posible la culminación exitosa del presente trabajo.

I N D I C E

Pág.

CAPITULO II Pág.

- CAPITULO III

Pág.

Pág.

Pág.

P R O L O G O

Los microprocesadores han tenido durante los últimos años un gran desarrollo, su influencia ha crecido en todos los campos, haciéndose presentes sus efectos en casi todas las industrias actuales. Cada día son más las aplicaciones de estos pequeños artefactos electrónicos y por lo tanto, es también mayor la necesidad de contar con facilidades de software y hardware que permitan su mejor aprovechamiento.

En el campo de software para microprocesadores para la resolución de problemas, el assembler es la vía más apropiada.

El assembler es un lenguaje de programación que utiliza códigos nmotécnicos para definir cada operador y permite etiquetar direcciones,. Usualmente tienen una relación directa cada instrucción en assembler con las tareas correspondientes al trabajo interno del computador.

La programación en otro lenguaje más complejo exige más capacidad de memoria de la que se tiene normal-

mos en el FORTRAN IV de la minicomputadora DM-250 de la General Automation.

CAPITULO V:

Utilización del trabajo realizado en la traducción de programas que presenten diversas facetas y posibilidades de manejo del assembler con que cuenta el sistema ensamblador desarrollado.

CAPITULO VI:

Comentarios acerca de lo conseguido en este trabajo, conclusiones, proyecciones y mejoras que pueden hacerse todavía para facilitar aún más la tarea de programación del usuario.

Además, se tiene apéndices que presentan material usado en la elaboración de la tesis: tablas de códigos, sets de instrucciones, descripción de la gramática desarrollada para el assembler con el que se ha trabajado, forma de uso del ensamblador a manera de Instructivo o manual de usuario y listados de los programas que constituyen el traductor.

C A P I T U L O I

INTRODUCCION GENERAL Y ALCANCES.

En este trabajo se estudia primero, en forma general, la base teórica de diseño de compiladores, luego se la enfoca al caso específico de ensambladores y, por último, se la aplica al diseño, programación e implementación de un assembler para la Motorola 6800, éste se corre en una minicomputadora DM-250 de la General Automation, produce código objeto y debe ser cargado manualmente en la microcomputadora para ser ejecutado.

Se utiliza el minicomputador de la General Automation porque tiene un sistema iteractivo que permite realizar eficiente y sencillamente la carga y edición de programas.

Antes de empezar a tratar de compiladores, lo que se hará en los capítulos siguientes, es necesario revisar algunos conceptos que serán de utilidad más adelante.

1.1 LENGUAJES DE PROGRAMACION DE ALTO NIVEL

En general un lenguaje de programación es una notación

lar registros y proyectar instrucciones que en la solución del problema planteado.

1.3 LENGUAJE DE MAQUINA.

El lenguaje de máquina de un ordenador no es más que los pasos elementales que debe dar la computadora, descritos en una secuencia de números binarios.

Virtualmente, nunca se programa en lenguaje de máquina. Programar en este código binario es bastante lento y existen muchas posibilidades de introducir errores. Es mucho más cómodo programar en assembler y luego traducir las instrucciones, por algún medio, a código de máquina.

1.4 DEFINICIÓN DE UN LENGUAJE DE PROGRAMACION.

Un lenguaje de programación está definido si se ha expresado claramente qué instrucciones son válidas y su significado.

Un programa puede ser considerado desde dos puntos de vista: el de la sintáxis y el de la semántica.

1.4.1 Sintáxis.

Un programa en cualquier lenguaje puede ser considerado como una secuencia de caracteres tomados de algún set o alfabeto de caracteres.

Las reglas que indican si una serie de caracteres pertenecen a un programa válido o no, constituyen lo que se conoce como sintáxis del lenguaje.

1.4.2 Semántica.

Cuando se conoce que un programa es válido, se necesita saber qué es lo que hace.

Se requiere saber qué significa un programa y si hace lo que el programador espera.

Las reglas que permiten hallar el significado de los programas, forman parte de la semántica del lenguaje. La semántica de un lenguaje de programación es mucho más compleja que su sintáxis.

Tanto la sintáxis como la semántica se estudiará con algún detalle en los Capítulos II y III.

CAPITULO II

BREVE ESTUDIO DE LA TEORIA DE DISEÑO DE COMPILADORES

2.1 INTRODUCCION

El objeto de este Capítulo es presentar en forma sumarizada y resumida las técnicas más importantes empleadas en la implementación de compiladores.

Se inicia con una breve mirada sobre los traductores los intérpretes y los ensambladores, los modelos matemáticos en que se basan los traductores y luego se procede a realizar una revisión de cómo se efectúa el proceso de compilación en sus diferentes etapas: análisis lexicográfico, análisis sintáctico, generación de código, organización de la tabla de símbolos, recuperación de errores, optimización de código, hasta estructurar un modelo general de un compilador.

2.2 TRADUCTORES, INTERPRETES

Un traductor es un programa cuya entrada es un programa escrito en un lenguaje fuente (programa fuente) y su salida es un programa equivalente en lenguaje objeto, a menudo código de máquina (Programa Objeto).

Si el lenguaje fuente es de alto nivel como FORTRAN,

la cual produce el código objeto se llama assembler residente o self-assembler. Si el assembler se lo pasa en otra computadora es un cross-assembler. Algunas veces el assembler residente está en memorias ROM de modo que no necesita ingresarlo cada vez que se desee utilizarlo. Sin embargo, su velocidad es baja y depende de la rapidez del microprocesador, por otra parte,usualmente el assembler requiere de periféricos con los que no se cuenta en un sistema microprocesador. Similar distinción se aplica a los self-compiladores y cross-compiladores.

En el desarrollo de software para microprocesadores por lo general se trabaja con cross-assemblers, pues resulta más conveniente y sencillo depurar y probar un programa en una computadora de mayor capacidad de memoria, mayor velocidad y más disponibilidad de periféricos que en el sistema microprocesador directamente, pues éste a menudo, debido a su aplicación, no dispone de sistemas operativos avanzados ni de los periféricos necesarios.

La salida del cross-ensamblador se la puede colocar en papel, cinta magnética u otro medio para que a continuación sea cargada en el procesador que lo va a ejecutar.

Por último, un método de ensamblaje rudimentario en el

Se le denomina máquina no en un sentido real, sino más bien como un modelo matemático, cuyas propiedades y comportamiento se pueden estudiar y que se pueden simular con un programa de computadora.

La máquina de estado finita es el más simple de los modelos de autómata. Son muy útiles en varias situaciones de diseño porque:

1. Tienen la capacidad de realizar tareas sencillas de compilación. Sobre todo en cuanto al diseño del analizador lexicográfico.

2. La simulación de una máquina de estado finita requiere un incremento de almacenamiento en memoria, pero simplifica los problemas de manejo de memoria.

3. Mediante unos pocos algoritmos se puede construir y simplificar máquinas de estado finitas.

Una máquina de estado finita está descrita por:

1. Un grupo de símbolos de entrada.
2. Un grupo finitos de estados
3. Una función de transición δ que asigna un nuevo

y que la máquina tiene una salida válida, entonces ningún nuevo estado será llamado. Cuando aparece un "NO" ó un 0 , la salida es no válida.

A continuación se va a ver un ejemplo explicativo de lo anterior:

Ej. 2.1

Consideremos el problema de reconocer 0's y 1's que contengan 2 consecutivos 1's dentro de la secuencia.

En la figura 2.1 se observa la tabla de transición para este problema donde C es el estado en el cual la secuencia 11 se detecta. El estado A es cuando ingresa un 0, el estado B cuando ingresa un 1.

Fig. 2.1.

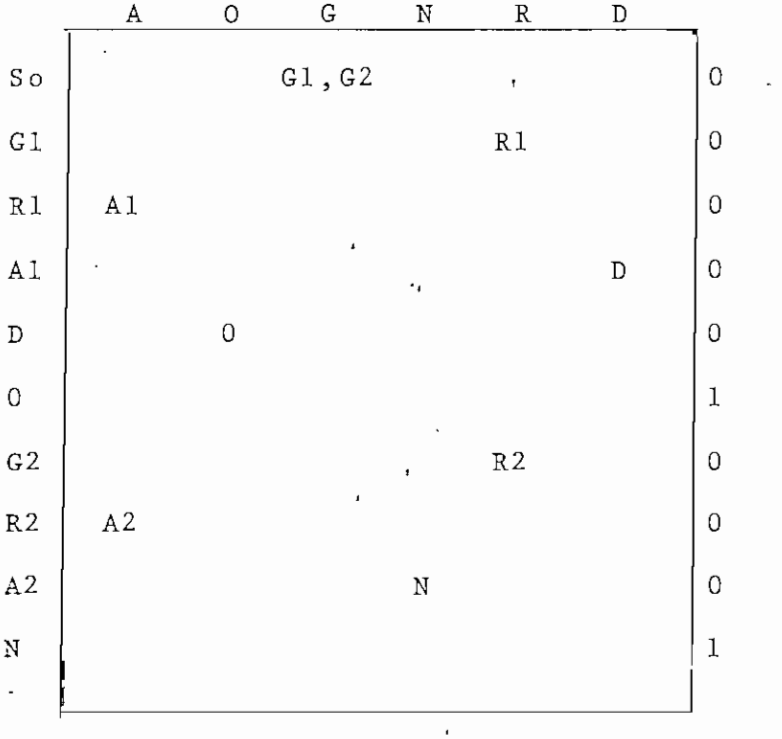
	Símbolos de entrada			
	0	1	⊣	marca de final
estados A	A	B	0	
B	A	C	0	salidas no válidas
C	C	C	1	salida válida

2.4.3 Conceptos útiles en máquinas de estado finitas

1. Secuencia nula : es una sucesión de caracteres blancos. Una máquina de estado finita acepta una secuencia nula, sí y solo sí su estado inicial es un estado aceptador.

So	estado inicial
G1	G en grado
R1	R en grado
A1	A en grado
DJ	D en grado
O	O en grado
G2	G en gran
R2	R en gran
A2	A en gran
N	N en gran

Fig. 2.2



Ej. 2.3

Supongamos un reconocedor de constantes válidas que acepte lo siguiente:.

1. Un dígito antes de un punto decimal
2. Punto decimal opcional
3. Digito después de punto decimal
4. Letra E
5. Signo de exponente
6. Dígito de exponente
7. Punto decimal

Fig. 2.3

	Dígito	E	.	Signo	
0	1		7		0
1	1	4	2		1
2	3	4			1
3	3	4			1
4	6			5	0
5	6				0
6	6				1
7	3				0

Se ve que la máquina no es reducida, pues los estados 2 y 3 son equivalentes. Se combina estos estados en uno solo que se llama $\overline{23}$. Se tiene entonces:

Fig. 2.4

	Dígito	E	. .	Signo	
0	1		7		0
1	1	4	$\overline{23}$		1
$\overline{23}$	$\overline{23}$	4			1
4	6			5	0
5	6				0
6	6				1
7	$\overline{23}$				0

2.4.5 Implementación de máquinas de estado finitas.

Antes o durante la implementación de máquinas de estado finitas hay 3 consideraciones básicas que se deben hacer:

a) utilizar las máquinas de estado finitas o no, resolviendo de acuerdo a la aplicación. Cuando se usan autómatas se logra rapidez en la ejecución a costa de un aumento en memoria de computador;

b) Decidir cómo manejar ciertos problemas de compilación recurrentes;

Fig. 2.5 Transliterator típico.

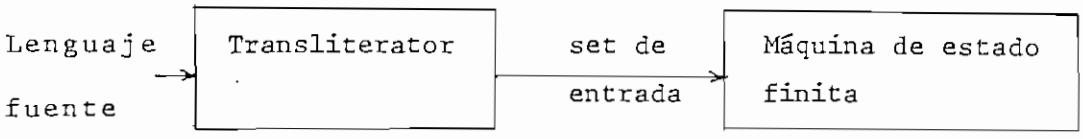


Tabla 2.1

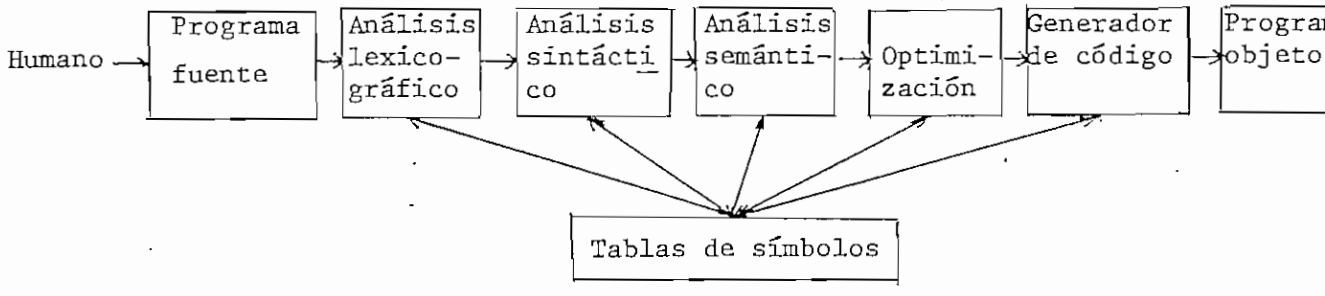
Carácter	Transliterator
a	(letra, a)
.	
.	
.	
Z	(letra, Z)
0	(dígito, 0)
.	
.	
.	
9	(dígito,9)
+	(operador,+)
.	
.	
.	
ETC	ETC

2.5 LA ESTRUCTURA DE UN COMPILADOR

La tarea de convertir un programa escrito en lenguaje fuente a otro equivalente en lenguaje objeto, es bastante compleja, por lo que es necesario dividir el pro-

ceso en varias etapas llamadas fases. Una fase es una operación lógicamente coherente que toma como entrada una representación del programa fuente y produce como salida otra representación. A continuación veamos cada una de estas fases en las que se puede separar el proceso de compilación. (Fig. 2.6)

Fig. 2.6 Modelo sumarizado de compilador



2.5.1 El explorador: Análisis Lexicográfico

El analizador Lexicográfico es un interfase entre el programa fuente y el compilador. En esta fase va leyéndose cada carácter de la sentencia en lenguaje fuente y se los va agrupando en una secuencia de unidades que son entidades lógicas llamadas TOKENS. Típicamente son Tokens los identificadores como nombres de variables tales como Yo, NOMB, llaves como DO ó GOTO, símbolos de operadores como $\angle$, =, +, -, etc. y símbolos de puntuación como paréntesis, comillas, punto y coma.

Cada token puede tener una parte de clase y algunos como constan-

tes y operadores tienen otra parte de valor.

Para encontrar los tokens, el analizador lexicográfico examina caracteres sucesivos en el programa fuente hasta determinar completamente el token.

Ej. 2.5

Supongamos una sentencia en FORTRAN analizada por el explorador:

IF (Z. LT. 2) GO TO 100

Se tiene 19 caracteres de entrada (incluídos espacios en blanco) el analizador los separa en los siguientes TOKENS:

IF

Z

LT

2

GO TO

El analizador lexicográfico y el paso siguiente el analizador sintáctico, están a menudo agrupados juntos en el mismo paso. En ese paso el analizador lexicográfico opera bajo el control del analizador sintáctico, éste último pregunta por el siguiente token cuando encuèntra uno. En el caso de que el token es un identifi-

cador u otro token con valor, se pasan tanto la clase como el valor al identificador, esta información se guarda en una biblioteca o tabla.

En el ejemplo visto,la tabla No. 2.2 estaría constituída como sigue:

Tabla 2.2

Token	parte de clase	parte de valor
IF	llave IF	no valor
Z	variable	puntero de entrada a la tabla de símbolos
LT	Operador	Valor '<'
2	constante	Valor 2
GOTO	Llave GO-TO	no valor
100	constante	Valor 100

2.5.2 El reconocedor: análisis sintáctico

Se llama parser o programa reconocedor a aquel que sólo reconoce

sentencias de la gramática en cuestión. El parser analiza que los tokens que ingresan a este bloque estén en un órden permitido en ese lenguaje fuente y además establece la jerarquía de las operaciones a ejecutarse por la identificación de los tokens que deben agruparse en nuevas entidades: los átomos, que al igual que los tokens pueden tener 2 partes: una de valor y otra de clase.

Veamos otro ejemplo.

Ej. 2.6

La expresión A/B * C+D

se interpreta en FORTRAN de la siguiente manera: divida A para B, luego multiplique por C y finalmente sume a D

Se agrupan los tokens así:
Divida (A, B, R1)
Multiplique (R1, C, R2)
Sume (R2, D, R3)

donde R1, R2 son resultados parciales, R3 resultado total de esa sentencia. R1, R2 y R3 son nombres de variables temporales.

Los tokens se han agrupado en 3 átomos, cuyas partes se muestran en la tabla No. 2.3, detallada a continuación:

Tabla 2.3

ATOMOS	Parte de clase	Parte de valor
Divida (A,B,R1)	División	Tres apuntadores a la tabla de símbolos para A, B y R1
Multiplique (R1,C,R2)	Multiplicación	Tres apuntadores a la tabla de símbolos para R1, C, R2
Sume (R2,D,R3)	Suma	Tres apuntadores a la tabla de símbolos para R2,D,R3

2.5.3 Análisis Semántico

Generalmente está dentro de otras fases del compilador. En el análisis semántico se halla el "significado" de los símbolos. Por ejemplo, la semántica de un identificador puede incluír su tipo y si es un arreglo sus dimensiones.

Una clase de proceso semántico incluye el llenar la tabla de símbolos con las propiedades de los identificadores, en forma individual como van ingresando al compilador. Otros procesos semánticos incluyen las actividades que dependen del tipo de datos. Por ejemplo, decisiones que dependen de los tipos de operandos son decisiones semánticas.

En algunos compiladores, ciertas actividades semánticas se colocan como una fase aparte, ubicada entre el análisis sintáctico y la generación de código.

2.5.4 Generación de código intermedio

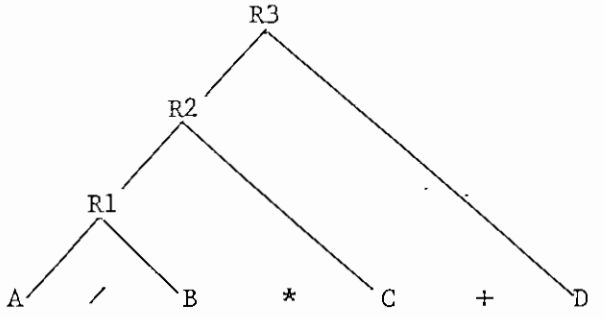
La salida del analizador sintáctico es una representación de un árbol de reconocimiento. En la generación de código intermedio se transforma este árbol en una representación intermedia del programa fuente.

Existen tipos de lenguajes intermedios, el más usado en compiladores es el llamado código de 3 direcciones. Veamos un ejemplo del uso de este lenguaje:

Ej. 2.7

En el caso visto anteriormente en el Ej. 2.6, se obtiene el siguiente árbol sintáctico:

Fig. 2.7



```
30 IF (I.LE.N) GO TO 50
   T1 = I * d2
    I = I + 1
   GO TO 30
50 CONTINUE
```

Realizando la aminoración de potencia; reemplazando una multiplicación por una suma que se ejecuta más rapidamente:

```
   I = A
  T2 = J * d2
  T1 = I * d2
  TI = 1 * d2
30 IF (I.LE.N) GO TO 50
  T1 = T1 + TI
   I =  I + 1
  GO TO 30
50 CONTINUE
```

De todas las formas se consigue que T1 final sea (A + X) * d2 con (A + X) $\leq$ N. Los dos pasos se los realiza generalmente a nivel de código intermedio. En el ejemplo se logra un programa que se ejecuta más rapidamente que el programa objeto que más obviamente corresponde al programa fuente en FORTRAN.

2. Optimización por Reducción.

Se realizan las operaciones cuyos operandos se conocen durante el tiempo de compilación, sustituyéndose ese

vía idéntica.

Normalmente un programador inserta operaciones redundantes para hacer más sencilla la lectura de un programa fuente, además aparecen operaciones redundantes cuando se trabaja con variables con subíndices. Al eliminar estas operaciones la ejecución será más rápida. Se va a ver un ejemplo de ésto a continuación:

Ej. 2.10

Sea la asignación

X (i,j): = X (i,j + 1)

En código intermedio:

(1) * i, d2

(2) + (1),j

(3) * i, d2

(4) + (3),j

(5) + (4),1

(6): =[X (2)],[X (5)]

Eliminando los pasos repetidos se tiene:

(1) * i,d2

(2) + (1), j

(3) + (2), 1

(4): =[X(2)], [X (3)]

2.5.6 Tablas de Símbolos

Un compilador requiere un conocimiento de toda la información que corresponde a los distintos nombres que aparecen a lo largo del programa fuente. Todos los atributos de los identificadores, tales como su tipo (real , entero, etc.), su forma (variable simple, estructura, etc.), su ubicación en memoria, entre otros, se los guarda en lo que se llama tabla de símbolos.

Una tabla de símbolos consta de 2 partes, una contiene el nombre del identificador y en la otra se guarda la información de que se dispone respecto a ese nombre (ver Fig. 2.8).

Las tablas de símbolos juegan un importante papel en el proceso de compilación, la información se ingresa durante el análisis sintáctico y lexicográfico y luego es utilizada en todas las otras fases de la compilación: en el análisis semántico, es decir el análisis de la consistencia de los nombres con sus declaraciones, durante la generación de código en la asignación de localidades de memoria que corresponden a cada nombre, en la detección y corrección de errores, tales como "variable no definida" ó "nombre de ora-

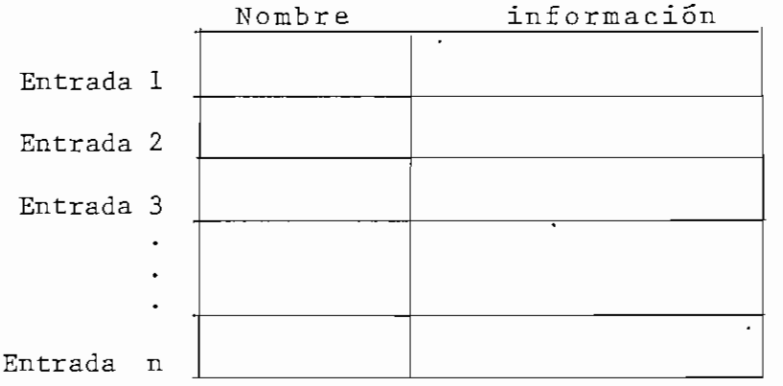
ción aparece más de 2 veces", e incluso en una etapa de optimización.

Las tablas deben estructurarse de modo que en ellas se pueda encontrar si un nombre ha sido o no ingresado y sacar en caso necesario la información respectiva, añadir o quitar un nombre de la tabla, añadir o cambiar la información correspondiente a un nombre.

En una sola tabla se puede guardar toda la información existente, pero usualmente no se trabaja de esa manera debido a la gran cantidad de información que se acumula para cada nombre, entonces las tablas se pueden separar en tablas de nombres, etiquetas, constantes y otros tipos de nombre.

Fig. 2.8

Organización de las tablas de símbolos



El compilador debe realizar búsquedas en las tablas antes de añadir un identificador nuevo a las tablas, por ello para optimizar el tiempo de compilación es necesario tomar en cuenta la organización más adecuada de las tablas para cada caso.

Una tabla puede irse conformando sumando nombres en el orden en el que van apareciendo. En este caso la búsqueda de un argumento en una tabla de n entradas, requiere de n/2 comparaciones. Este método es ideal en el caso de que n no sea muy grande. Si es que el número de entradas es mayor que 100 ó más (dependiendo de la velocidad del ordenador en el que se vaya a compilar el programa fuente), se puede clasificar los argumentos de las tablas usando algún orden lógico y realizar las búsquedas con alguno de los métodos que se van a explicar:

1. Búsqueda binaria o logarítmica.

Si se ordena los argumentos de acuerdo a algún método, como por ejemplo por el orden alfabético de los nombres, se puede realizar la búsqueda comparando el símbolo con la entrada del medio (en una lista de n entradas: n/2), si el número de órden es mayor que el guar-

dado en esa entrada, se protede a buscar en la parte inferior repitiendo el proceso con esa mitad de la lista, si el símbolo es menor se toma la porción superior. Este proceso se repite hasta hallarse el argumento buscado o hasta que se terminen las comparaciones posibles.

Para una lista de n entradas el máximo número de comparaciones que se va a realizar es $1 + \log._2 n$.

2. Organización de la tabla de símbolos por árboles.

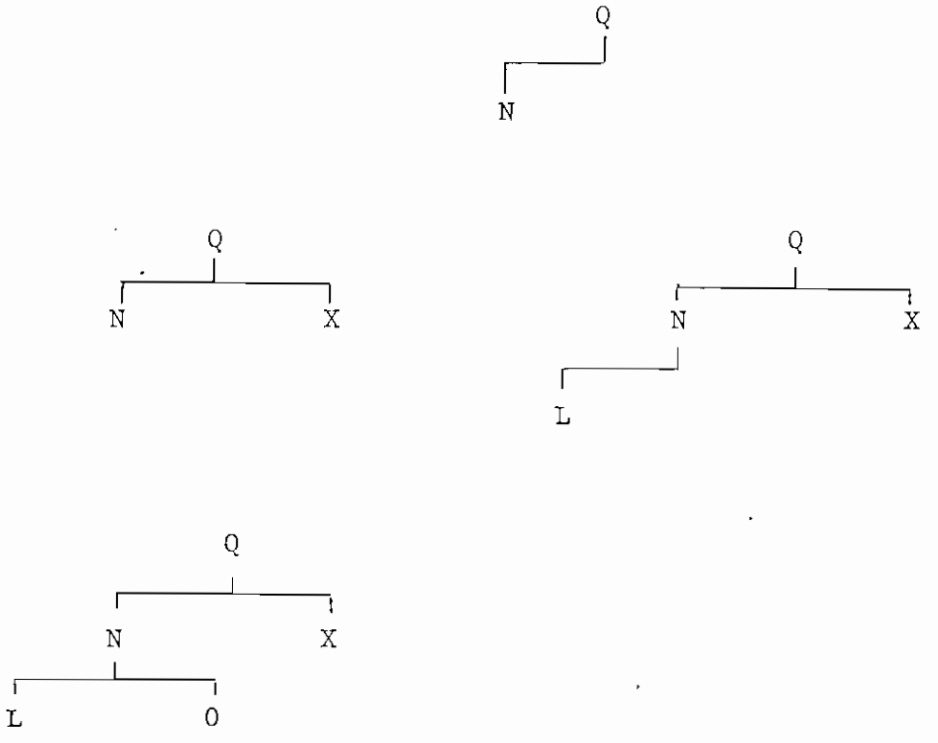
Consiste en ordenar las entradas en un árbol binario que tiene 2 nudos. Esto se realiza añadiendo 2 campos a cada registro, uno para la rama baja y otro para la alta. Se explicará la formación de un árbol con un ejemplo:

Ej. 2.11

Sea una tabla con una entrada para el identificador Q, luego se añade el identificador N, se dibuja una rama hacia la izquierda, pues por orden alfafético $N \measuredangle Q$, a continuación entra el identificador X, como $X \measuredangle Q$ se dibuja una rama hacia la derecha. Ingresa el identificador $L \measuredangle Q$, por lo que siguiendo la rama de la iz-

quierda llegamos a N, L $\measuredangle$ N, por lo que se dibuja una rama a la izquierda de N. Finalmente, se añade el identificador O, O $\measuredangle$ Q, por lo que tomamos la rama de la izquierda, O $\diagdown$ N, entonces dibujamos una rama a la derecha. De igual manera se colocan los identificadores M y P mostrados.

Fig. 2.9



El número de comparaciones necesarias depende del orden de llegada de los identificadores.

3. Búsqueda por Direccionamiento Hashing.

Este es un método más eficiente que los anteriores. Se basa en convertir símbolos a índices de entradas en la tabla. El índice se obtiene realizando una operación aritmética ó lógica sencilla con el símbolo. Esta técnica tiene muchas variaciones, aquí se va a considerar una variante simple.

El esquema básico de hashing consiste en 2 tablas, una para guardar los índices y otra para guardar la información. (Fig. 2.10)

Los índices almacenados en la primera tabla son punteros de la información guardada en la segunda. Para hallar cada nombre en la tabla de símbolos se aplica una función hash tal que h (nombre) sea un entero entre 0 y K-1 (para una tabla de K nombres). Este entero está en la tabla de índices. Cuando se busca un nombre se computa h(nombre) y se busca en la tabla de índices y con este puntero se saca la información de la tabla de información. Cuando se desea ingresar un nombre, se crea para él un registro en la tabla de información y se enlaza este registro con el puntero almacenado en la tabla de índices.

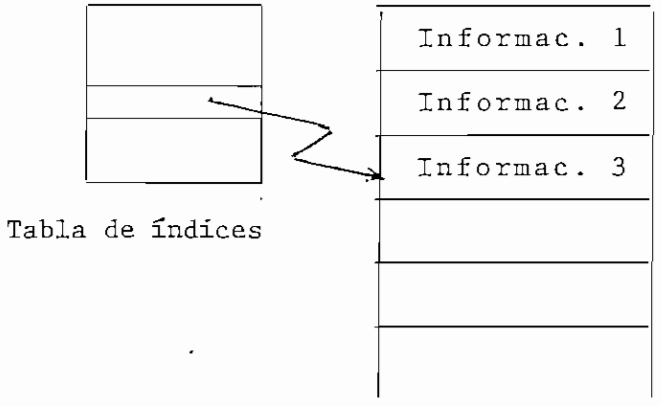
Hay dos criterios que se deben tomar en cuenta para realizar un hashing eficiente:

a- La función h debe distribuir en forma uniforme los índices a lo largo de la tabla.

b- h debe ser fácil de procesar.

Con el criterio a- se tiende a evitar el problema de la colisión que aparece cuando dos símbolos al aplicar la función h dan el mismo índice. Como es obvio, sólo podemos poner un símbolo en cada entrada, por lo que se debe ubicar el segundo en otro registro. Si los índices calculados se distribuen de una manera uniforme en la tabla, se logrará que existan menos colisiones.

Fig. 2.10

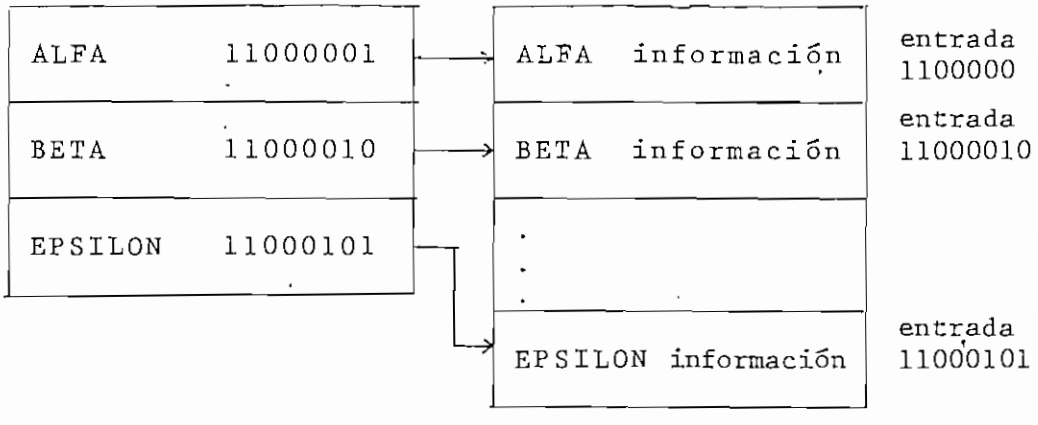


Ej. 2.12

Sea la función h tal que tome la representación bina-

ria del primer carácter del símbolo. Para el nombre ALFA se tomará la representación binaria de A,por ejemplo 11000001. Para el nombre BETA de la misma forma se tomará la representación binaria de B, por ejemplo 11000010, para el nombre EPSILON la representación de E 11000101 y así sucesivamente.

Fig. 2.12



Este hashing no es óptimo porque todos los nombres que empiecen con el mismo carácter tendrán igual índice y producirán demasiadas colisiones, pero es útil para aclarar la forma de este tipo de direccionamiento.

2.5.7 Detección y manejo de errores.

Una parte esencial de un compilador es aquella que se encarga de la detección y manejo de errores.

Hay distintas formas de manipular los errores que varían en complejidad y van desde la simple detección a la recuperación, la separación ó la corrección de los errores. Un compilador con un sistema simple de tratamiento de errores se detendrá cuando detecte el primero, otro compilador más completo puede intentar reparar el error reemplazando una entrada equivocada por otra correcta y resumir la compilación en el punto en que se detectó el error, un compilador más sofisticado puede tratar de corregir los errores "adivinando" las intenciones del programador. Este último tipo de compilador está expuesto a introducir más errores de los que corrige, por cuanto muchas veces lo que el programador deseaba hacer resulta difícil de predecir con lo que se malgasta el tiempo de compilación, además de complicar innecesariamente el proceso.

La detección de errores puede realizarse en cualquier etapa del proceso de compilación. Por ejemplo:

- En el analizador lexicográfico al detectar un token equivocado supongamos una oración en FORTRAN.

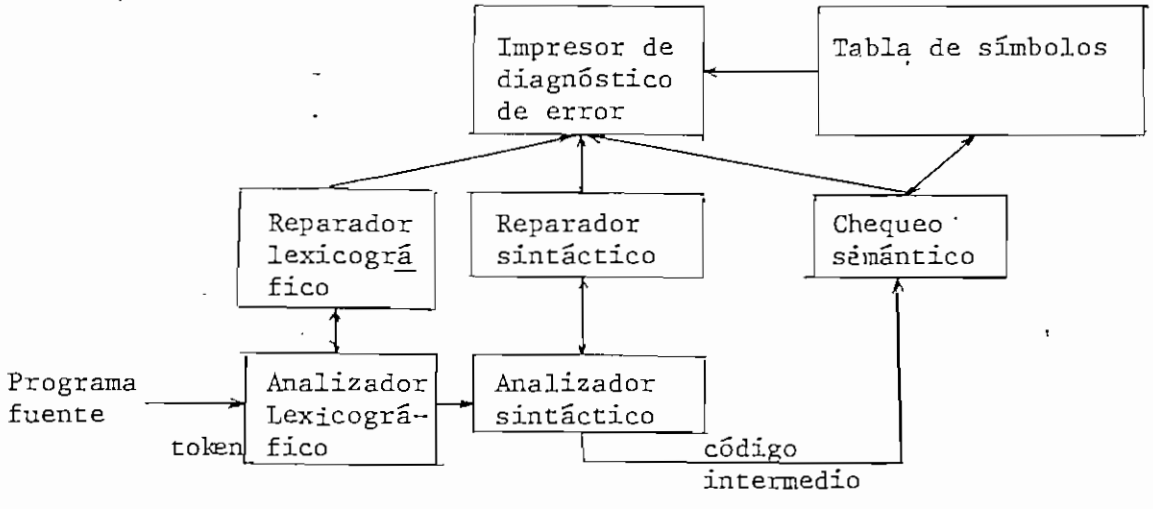
IF (5.EQ.MAX) GOT 400

Los tokens hallados serán IF, 5, EQ, MAX, los parén-

Para poder seguir realizando la compilación, una vez detectado el problema, el computador debe modificar la entrada incorrecta, de modo que el proceso continúe y se sigan detectando errores existentes más adelante en el programa.

Un esquema de ésto se muestra en la Fig. 2.12

Fig. 2.12



Se ha discutido ya, en forma breve, la detección de errores Lexicográficos y sintácticos, veamos ahora algo sobre la detección de errores en el chequeo semántico.

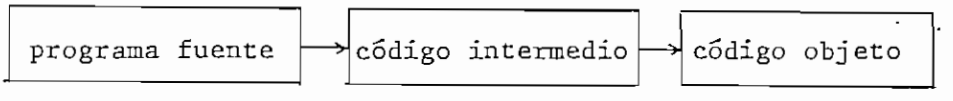
Errores semánticos son aquellos motivados por el uso incorrecto de identificadores y expresiones. En esta

fase la recuperación de errores consiste en reemplazar un identificador o expresión incorrecta, por otro sin error, insertando una nueva entrada en la tabla de símbolos con los atributos determinados por el programa hasta el momento de la detección del error y cambiando el apuntador interno para que señale esa nueva entrada. Un ejemplo típico de esta clase de error, son los identificadores no declarados o declarados múltiples veces.

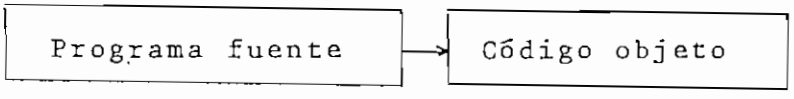
Distintos tipos de error pueden ser detectados tanto durante el tiempo de compilación, ejemplos de lo cual ya hemos mencionado, como en el de corrida. Una detección en tiempo de corrida es el chequeo de valores en una oración CASE (en PL/I) o un GOTO calculado que cause un salto a una localidad de memoria inexistente.

2.5.8 Generación de código.

En la fase de generación de código se convierte el código intermedio en una secuencia de instrucciones de máquina. El proceso a producirse es:



Las rutinas semánticas generan el código intermedio, pero también pueden saltarse este paso y generar directamente el código de salida del compilador. Esto es, se puede compilar sin producir código intermedio. Entonces se tendrá:



Esto complica mucho más las rutinas semánticas, sobre todo cuando se trata de un traductor de lenguaje de alto nivel, pero también hace más rápida la compilación.

De todas formas se asume que previo a la generación de código objeto, el programa fuente ha atravezado todas las etapas anteriores, ha sido explorado, reconocidas sus sentencias y, en forma opcional, traducido a un código intermedio. El análisis semántico también debe haberse realizado insertándose, por ejemplo, las operaciones de conversión de tipos de variables y detectándose los errores semánticos (por ejmplo el tener índices no enteros en FORTRAN).

Antes de la generación de código objeto, cada operador y operando, es conocido con todos sus atributos.

Sin embargo, no es indispensable que haya existido en el compilador una fase de optimización a nivel de lenguaje fuente.

Uno de los problemas al realizar un sistema de generación de código en la compilación de un lenguaje de alto nivel, es el de ahorrar instrucciones de código objeto. Comunmente una conversión directa de código intermedio a código de salida, produce un programa objeto que contiene instrucciones redundantes sobre todo de carga y almacenaje que tienden a convertir ineficiente al programa. Para evitar ésto, un buen generador de código debe rastrear el contenido de los registros durante el tiempo de corrida. Conocidas las cantidades existentes en los registros, el generador de código produce instrucciones de carga y almacenaje sólo cuando es necesario.

La generación de código para ensambladores, usualmente no realiza esta optimización porque generalmente se desea que la correspondencia entre instrucciones en lenguaje ensamblador e instrucciones en código de máquina, sea de uno a uno.

El código objeto generado en esta etapa de compilación puede ser de una de estas tres formas:

1. Instrucciones absolutas, situadas en posiciones fijas que el compilador ejecuta entonces inmediatamente.

   Esta posibilidad es la más eficiente desde el punto de vista del tiempo, pero no brinda mucha flexibilidad porque no se pueden precompilar varios subprogramas por separado para luego montarlos juntos y ejecutarlos, pues todos se deben compilar al mismo tiempo.

   Computadores que trabajan de esta forma son, por ejemplo, WATFOR y ALGOL.

2. Código objeto generado para un lenguaje de alto nivel en lenguaje assembler que obviamente no puede ser ejecutado directamente, sino que debe ser procesado previamente por un ensamblador.

   Esta es la más simple de las alternativas porque no se tiene que producir las instrucciones como series de bits, sino expresiones simbólicas o incluso llamadas a macros, y las definiciones que soporte el ensamblador.

   Esta forma es casi siempre la peor, pues añade al proceso de traducir un programa, un paso adicional que a veces consume tanto tiempo como la compilación misma.

3. Código objeto ó binario, programa en lenguaje de máquina que se almacena en memoria auxiliar. Debe montarse con otros subprogramas y luego se carga para ejecutarse.

   El código objeto puede también ser de dos clases:

   - Relocatable si puede cargarse en cualquier lugar de la memoria.
   - Absoluto si debe cargarse en unas posiciones fijas.

   En el primer caso, cada instrucción o valor que se refiera a una posición de memoria dentro del programa, debe poder relocalizarse de manera que se pueda cambiar esa referencia a memoria, sin importar donde se carga el programa.

   En algunas máquinas de "tiempo compartido" no existe diferencia entre un módulo absoluto y uno relocalizable, porque siempre se realiza reubicaciones en las localidades de memoria y esos cambios se los hace por hardware.

2.6 BLOQUES Y PASADAS DE UN MODELO DE COMPILADOR.

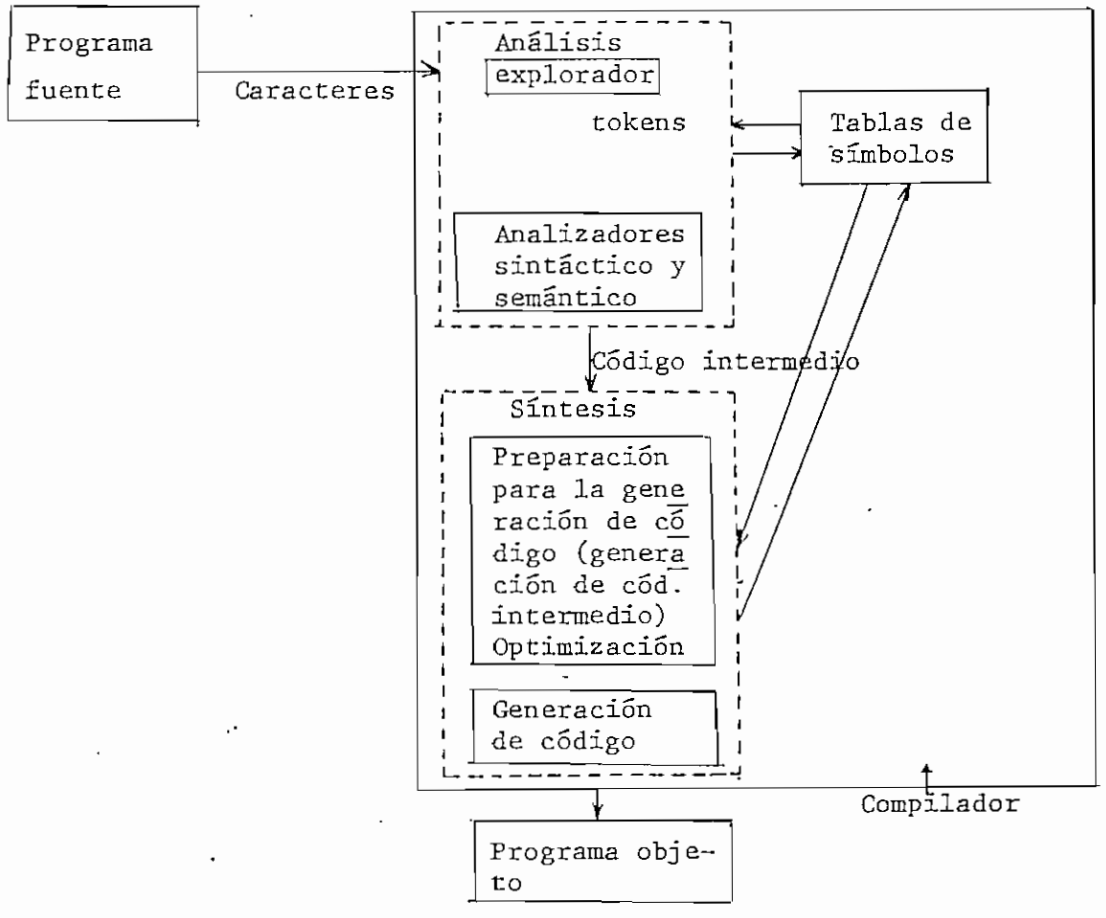
Un compilador debe realizar un análisis del programa fuente y luego una síntesis del programa objeto. El compilador descompone el programa fuente en sus componentes básicos y entonces construye trozos de pro-

gramas objeto a partir de ellos. En la fase de análisis el compilador crea tablas de símbolos que serán de utilidad tanto en el análisis como en la síntesis.

2.6.1 Bloques de un compilador

Como ya se dijo anteriormente, el proceso de compilación se subdivide en fases o partes lógicas, en las cuales se realiza algún tipo de transformación de sus entradas. Se ha mencionado ya en forma individual las partes que constituyen un compilador, veamos en un diagrama de bloques la forma como éstas se interrelacionan.

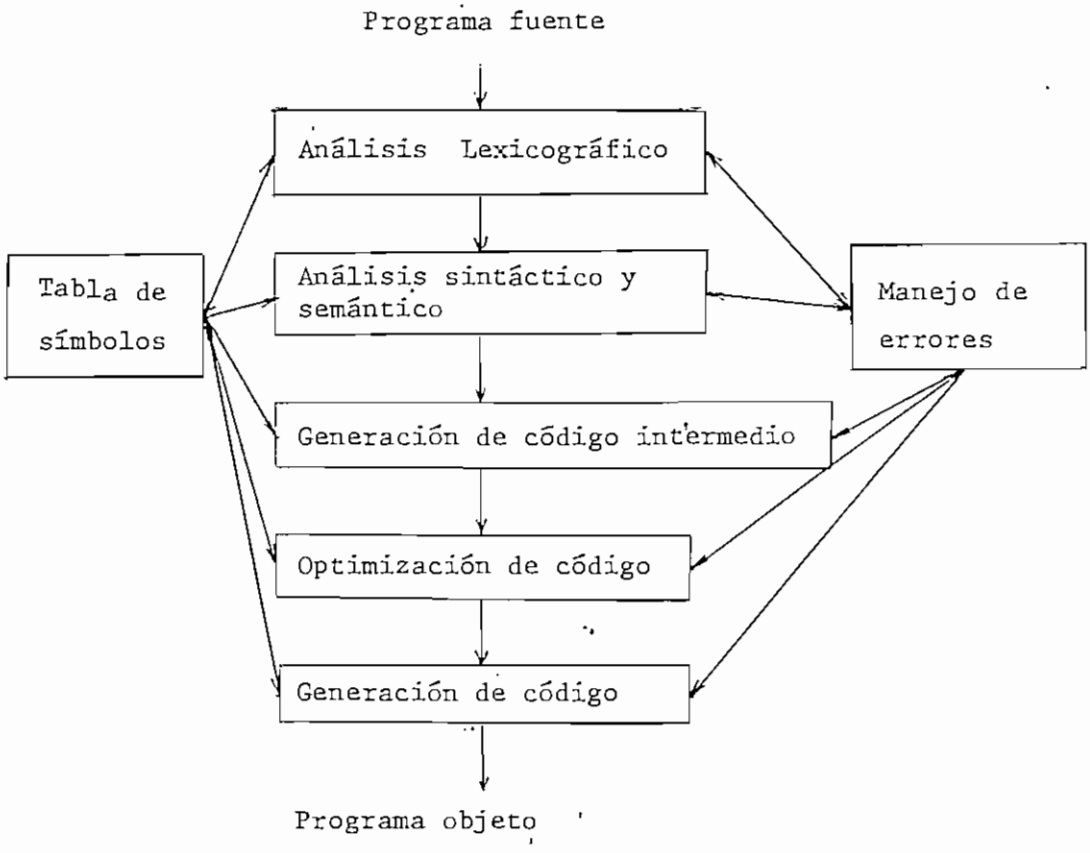
Fig. 2.13 Partes lógicas de un compilador



2.6.2 Pasos de un compilador

La Fig. 2.13 muestra las partes lógicas de un compilador y no su relación temporal. Las fases de un compilador en un orden sucesivo se muestra en la Fig. 2.14.

Fig. 2.14



Ya se ha visto la función de cada una de esas fases. Los procesos señalados en la Fig. 2.14 se pueden ejecutar en el orden mostrado ó en forma paralela, entrelazada. En la implementación de un compilador se com-

binan una ó más fases en módulos llamados pasos. De acuerdo a cómo se efectúen las fases se tiene compiladores de uno o varios pasos.

Para decidirse por un compilador de 1 o más pasos que resulte el más apropiado para ciertas aplicaciones, hay algunos criterios que se deben considerar:

- Cantidad de memoria disponible. Un compilador de un paso requiere mayor asignación de memoria que compiladores multi-pasos. En un compilador de varios pasos se puede ocupar el mismo espacio de memoria usado por el paso anterior en cada paso siguiente.

- La estructura del lenguaje fuente. Ciertos lenguajes necesitan al menos 2 pasos para generar código con más facilidad. Por ejemplo lenguajes como PL/I requieren qùe la declaración de un nombre se produzca después del uso del mismo y el código no puede ser generado convenientemente hasta que la declaración se realice.

- La rapidez que se necesite en el compilador

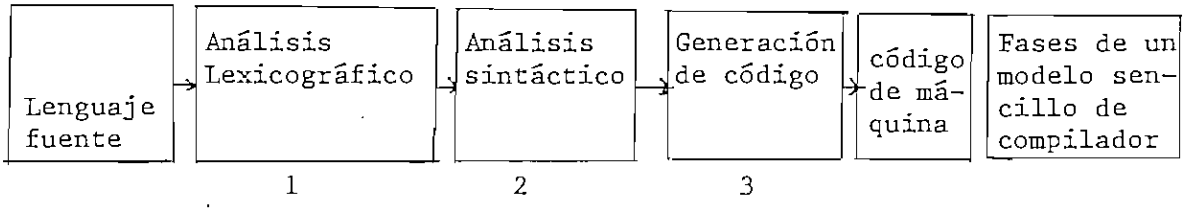
- La rapidez que se necesite en el programa objeto

- Facilidades de depuración indispensables.

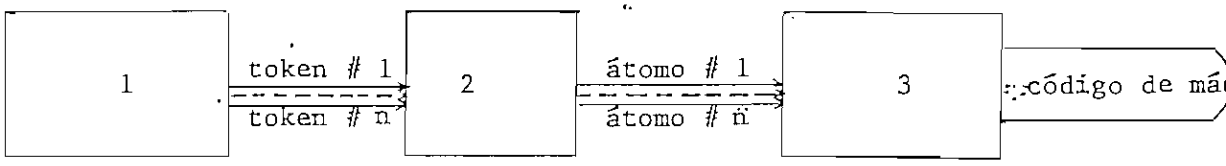
Ej. 2.14

Un modelo reducido de compilador puede ser organizado en pasos en distintas formas, veamos:

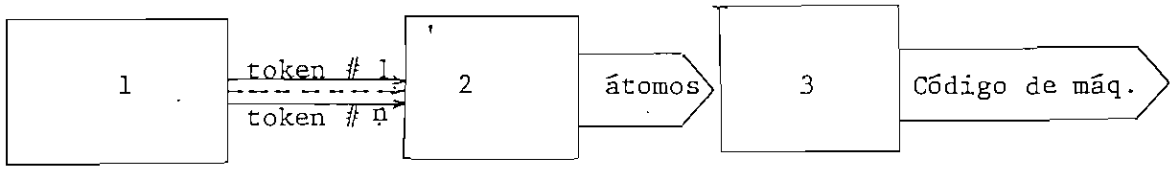
Fig. 2.15



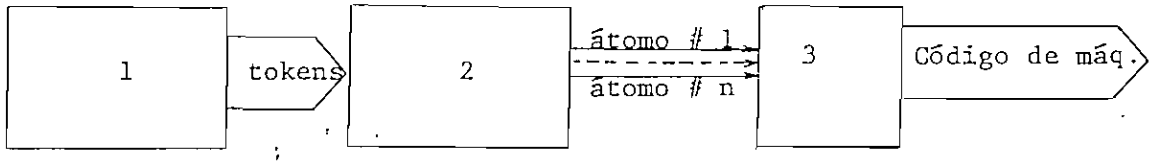
a) Compilador de un paso



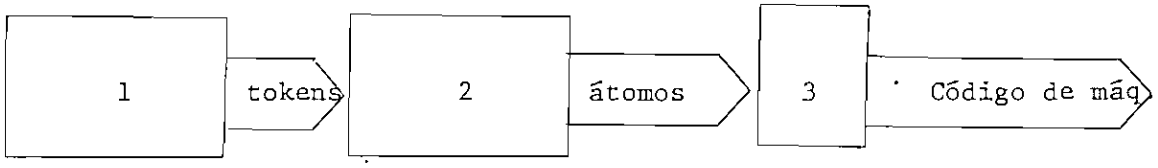
b) Compilador de dos pasos



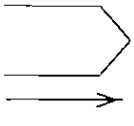
c) Compilador de dos pasos



d) Compilador de tres pasos



Donde:



indica una transferencia paralela

indica una transferencia serial

Una organización en pasos puede introducir alguna sobrecarga extra, porque cada interfase entre pasos requiere que toda una secuencia de símbolos de salida sea salvada. Sin embargo, muchas veces las ventajas que este tipo de estructura brinda, son mayores que las desventajas.

2.6.3 Assemblers de uno ó más pasadas.

La mayoría de ensambladores son de dos pasadas. Durante la primera pasada se crea la tabla de símbolos y se recogen todas las definiciones; en la segunda pasada, se traduce el programa con la ayuda de la información recolectada en el primer paso. Este tipo de assembler es bastante versátil, los símbolos pueden definirse en cualquier sitio a lo largo del programa, porque de todas formas se va a leer completamente todo el programa durante la primera pasada. Luego de esta pasada puede no requerirse una segunda leída si se almacena la información en cinta ó disco; si no existe esta posibilidad, el ensamblador de dos pasadas debe leer el programa dos veces a partir de cinta de papel, tarjetas o terminal.

Un assembler con salida a cinta de papel puede reque-

una tercera pasada.

Ej. 2.15

Supongamos el siguiente programa para la M6800:

Fracción de programa que suma números de 8 bits. Por ejemplo los contenidos de las localidades de memoria 0050 y 0051 y ubiquemos el resultado en la localidad de memoria TEMP, cuya dirección es 0040. Las instrucciones tienen además referencias usadas en otras porciones de programa, a fin de ilustrar el uso de la tabla de símbolos.

Sea el contenido de esas localidades:

NOTA: todas son direcciones hexadecimales

(0050) = 48

(0051) = 2D

Resultado: (0040) = 75

Fracción de programa en aseembler:

```
TEMP  EQU   $40
SUM   LDAA  $50   carge el primer
                  operando
      ADDA  $51   sume el segundo
                  operando
ST    STAA  TEMP  Guarde el resul-
                  tado
```

Fig. 2.16

Paso 1: Creación de la tabla de símbolos.

Programa				Tabla de símbolos	
		JUMP	THRU	THRU	205
123	SUM	LDAA	$50	SUM	123
124		ADDA	$51	ST	125
125	ST	STAA	TEMP	TEMP	$40

La etiqueta THRU aparece después en el programa y tiene como diferencia la dirección de memoria 205 hexadecimal.

Paso 2: Uso de la tabla de símbolos

Programa				Programa no simbólico	
		JUMP	THRU	JUMP	205
123	SUM	LDAA	$50	LDAA	$50
124		ADDA	$51	ADDA	$51
125	ST	STAA	TEMP	STAA	$40

Un ensamblador de una pasada tiene la ventaja de ser más rápido, porque el programa debe ser leído sólo una vez, tiene, en cambio, la desventaja de que sus reglas de direccionamiento, uso de nombres y localización de almacenamientos, son bastante estrictas.

De una manera general se puede afirmar que aunque algo

menos rápido, el compilador de dos pasadas ofrece mayores facilidades al programador que el compilador de una pasada.

2.7 IMPLEMENTACION EN TIEMPO REAL

Usualmente el problema del diseño del compilador está especificado en una manera incompleta, sobre todo en cuanto a los detalles de la salida producida por el compilador. Generalmente sólo se pide al diseñador que la salida debe corresponder al significado del lenguaje y debe satisfacer ciertos requerimientos de velocidad de ejecución o memoria de acuerdo al computador en que se va a correr y/o ejecutar.

Por lo tanto, el primer paso en el diseño es planear cómo debe ser la salida del compilador. Esto incluye el tipo de estructura de datos y mecanismos de control que existirán en el tiempo de ejecución para implementar los diferentes aspectos del lenguaje. Por ejemplo, cómo se cargarán los arreglos en memoria,qué procedimientos serán llamados, si serán permitidos procedimientos recursivos y cómo serán manejados.

El grupo de estructuras de datos y mecanismos de con-

2.7.2 Administración de memoria.

La memoria se organiza de formas diferentes en los distintos lenguajes de programación. Hay organizaciones sencillas para lenguajes como assembler, FORTRAN,. otras más complicadas para lenguajes tipo ALGOL, y las más complicadas en lenguajes que como el PL/I tienen estructuras, variables apuntadores y sentencias ALLOCATE y FREE.

En algunos sistemas la memoria se asigna pero nunca se libera de una forma explícita, cuando no hay más memoria disponible, el sistema llama a un "recogemigas" que halla las posiciones que no se están empleando y las libera. Otros sistemas, como se mencionó anteriormente, requieren asignación dinámica al azar, de memoria. Unos más, necesitan asignaciones estáticas.

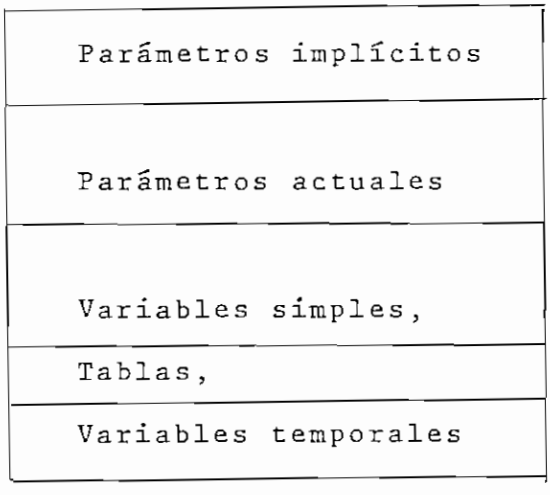
El estudio de la administración de memoria es bastante especializado para cada lenguaje y es materia que no está dentro de los objetivos de este trabajo, por ello sólo se presentará un ejemplo breve sobre la organización de la memoria en un lenguaje; el FORTRAN.

Ej. 2.16

En FORTRAN no se admite la asignación dinámica de memoria (todos los límites dados en la sentencia DIMENSION deben ser constantes), y por lo tanto no se requiere administración de memoria en tiempo de ejecución. El compilador puede asignar memoria a las tablas en un área de datos de longitud fija.

Veamos como ejemplo un área de datos estáticos típica del FORTRAN. Los parámetros implícitos incluyen la dirección de retorno y otras impuestas por el diseño del computador.

Fig. 2.17



En este caso una llamada a una subrutina consta de las siguientes operaciones:

CAPITULO III

PARTICULARIZACION DE LA TEORIA DEL CAPITULO II AL CASO DE UN TRADUCTOR CROSS-ASSEMBLER Y HACIENDO REFERENCIA AL MICROPROCESADOR 6800 DE LA MOTOROLA.

3.1 INTRODUCCION

Como ya se dijo anteriormente, un cross-ensamblador es aquel que realiza el ensamblaje de un programa en assembler, no en el mismo aparato que lo va a ejecutar, sino en una computadora de bastante más capacidad en memoria y mayor disponibilidad en cuanto a periféricos, sistemas operativos, editores y otras facilidades tanto de software como de hardware. De esta manera el cross-compilador puede realizar depuraciones y optimizaciones de programas en una forma mucho más rápida y eficaz, y una vez que se obtienen los programas ensamblados, simplemente se los debe cargar en la máquina a la cual corresponde el código objeto obtenido para que sean ejecutadas.

En este capítulo se trata de aplicar toda la teoría vista previamente en el Capítulo II, para dar una

- Una lista de errores de ensamblaje.
- Una tabla de símbolos con los significados de todos los nombres usados en los programas.
- Una tabla de referencias con una lista de nombres y todas las instrucciones que los usan.;
- Una lista de subrutinas o macros y sus longitudes.

El cross-assembler debe colocar el código objeto que produce en un medio apropiado, de modo que pueda ser cargado manualmente en el procesador que va a ejecutar dicho código de máquina. Usualmente la salida del cross-assembler se produce en papel, aunque si se cuenta con las facilidades necesarias, también puede ser ubicada en cinta magnética o en cualquier otro mecanismo de salida que permita recuperar el programa objeto para introducirlo en la memoria del ordenador correspondiente.

3.3 ORGANIZACION DE UN CROSS-ASSEMBLER.

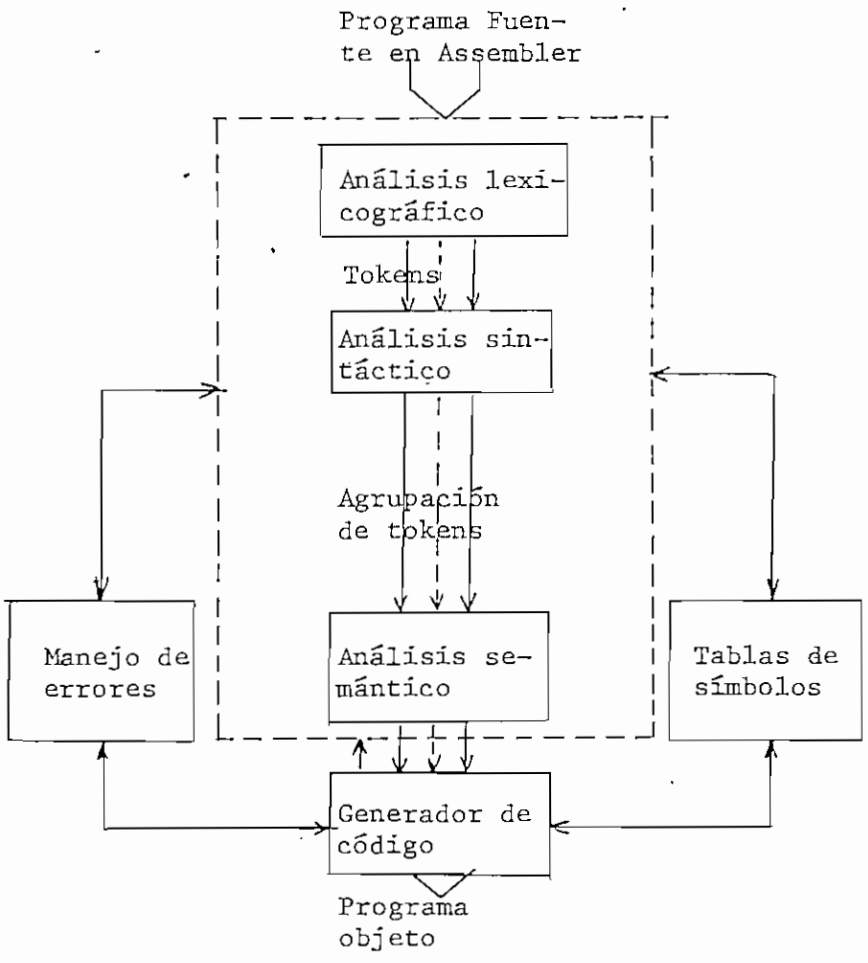
En el diseño de un cross-ensamblador se pueden aplicar de una manera general las técnicas estudiadas en el capítulo anterior. Sin embargo, hay que considerar que un ensamblador, por ser un lenguaje más próximo al de máquina, es más sencillo que un compilador

de un lenguaje más complejo, como por ejemplo: FORTRAN o COBOL.

De todas maneras, al programar un assembler, se puede observar que, según los alcances de que éste disponga, tiene más o menos los mismos bloques de un compilador promedio, aunque cada fase es más sencilla en un ensamblador que en un traductor de alto nivel.

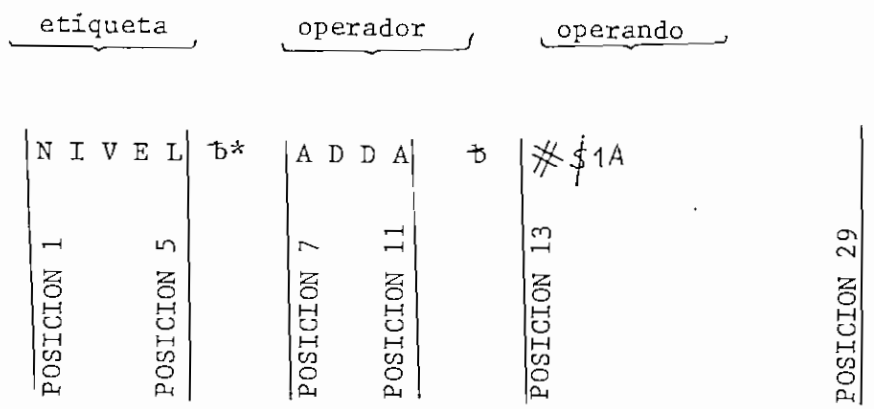
En la Fig. 3.1 se observa la estructura de un assembler típico.

Fig. 3.1 ESTRUCTURA DE UN MODULO ASSEMBLER TIPICO



Ej. 3.1

Sea un assembler con el siguiente formato:



Los tokens ó entidades lógicas vienen ya determinados por la posición que ocupan en el registro de entrada.

En este caso los tokens son: NIVEL-etiqueta, ADDA-operador, $18-operando.

En el análisis del operando si es necesario un buscador para dividirlo en sus partes constitutivas. A continuación un ejemplo.

Ej. 3.2

Sea un registro de entrada en assembler, suponiendo el mismo formato del ejemplo anterior.

b significa espacios en blanco

N I V E L		A D D A		X,10	
POSICION 1	POSICION 5	POSICION 7	POSICION 11	POSICION 13	POSICION 29

En este ejemplo el buscador deberá actuar en el operando y descomponerlo en los siguientes tokens: X - identificador de direccionamiento; "," - separador; 10 - constante.

3.3.2 Analizadores sintáctico y semántico.

Generalmente el analizador sintáctico o "reconocedor" de expresiones válidas, se confunde con el discriminador semántico que de acuerdo a la disposición de estos identificadores encuentra su significado. Esto sucede porque las dos fases trabajan muy íntimamente ligadas.

En el análisis sintáctico, la mayoría de las ocasiones no se requiere agrupar los tokens en átomos que establezcan la jerarquía de las operaciones a ejecutarse porque las operaciones en assembler se van realizando en el orden determinado por las instrucciones, a menos que se traten de saltos (JUMP,BRANCH.), caso en el que sí se forman átomos que establecen la secuencia de las

operaciones que se deben ejecutar. Es decir que en assembler, el reconocedor, usualmente, sólo analiza las sentencias, acepta como válidas las que se rijen por la gramática del lenguaje y declara inválidas las que no lo hacen. Esto se muestra en el ejemplo 3.3.

Ej. 3.3

Se tiene las siguientes instrucciones en un programa para la M6800 que encuentra el mayor de dos números:

```
        LDAA    $40     toma el primer operando
        CMPA    $41
        BHI     INIVE?  reemplaza con el segundo operando si el
        LDAA    $41     segundo es mayor
INIVE?  STAA    $42     guarda el operando más grande
        SWI             interrupción por software
```

El programa luego de ser descompuesto en tokens pasa al analizador sintáctico.

El reconocedor concluye que todos los identificadores son válidos.

No hace falta un árbol sintáctico que determine el or-

den de las operaciones, porque todas se deben realizar en el mismo orden en que se encuentra el programa.

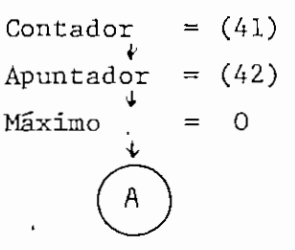
Ej. 3.4

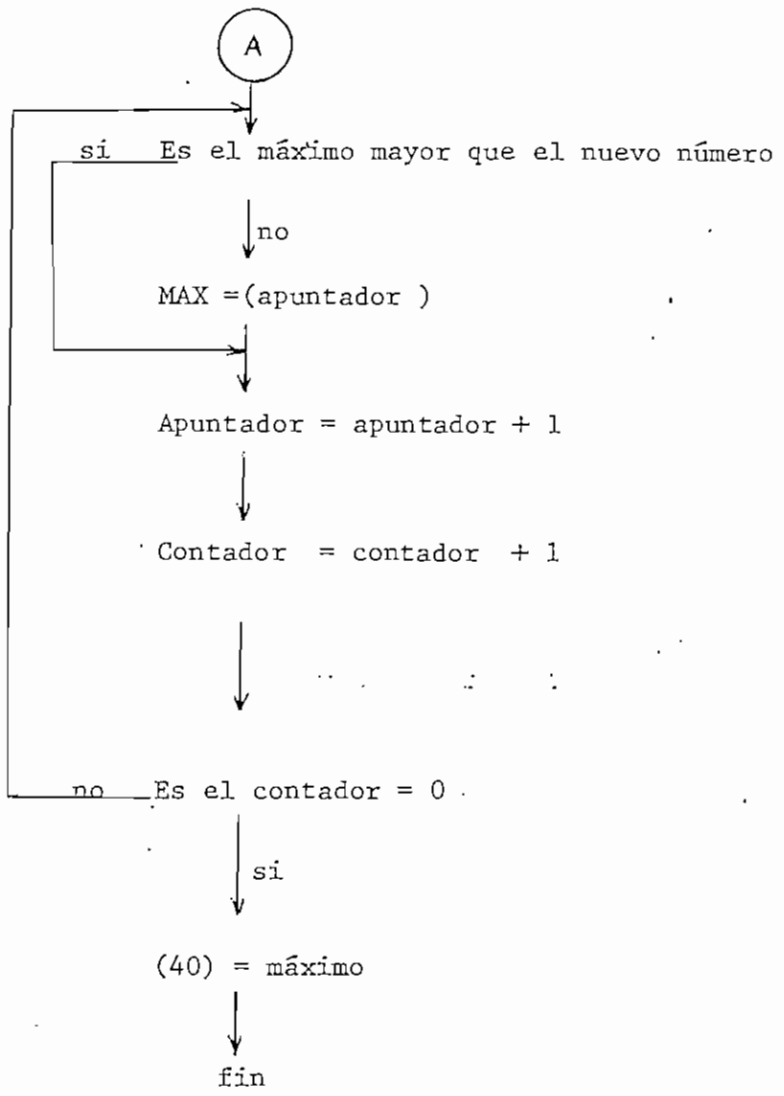
En este ejemplo se verá la formación de un árbol sintáctico ante la presencia de saltos.

Sea un programa escrito en el assembler de la M6800 que encuentra el máximo número de un bloque de datos.

```
        LDAB    $41     Contador
        CLRA            máximo = 0
        LDX     #$42    puntero en posición inicial
MAX     CMPA    X       es el máximo mayor que el numero?
        BCC     NOCAM   Si, se mantiene el máximo
        LDAA    X       No, reemplace el máximo con el
                        nuevo número?
NOCAM   INX
        DECB
        BNE     MAX
        STAA    $40     guarde el máximo
        SWI
```

El árbol sintáctico





Muchas veces los analizadores lexicográfico, sintáctico y semántico se encuentran dentro de los mismos programas, por esta razón, en la Fig. 3.1, estas tres fases se colocan en un solo bloque (línea de puntos).

3.3.3 Generación de lenguaje intermedio.

Esta fase generalmente se omite en la programación de un assembler, debido a que el árbol sintáctico que es-

M6800.

Suma de datos:

# inst.	Dirección memoria				
1	0100		CLRA		Suma = cero
2	0101		LDAB	$41	contador = longitud del arreglo
3	0103		LDX	#$42	apunta el principio del arreglo
4	0106	SUMA	ADDA	X	suma = suma + (apuntador)
5	0108		INX		apuntador = apuntador + 1
6	0109		DCB		contador = contador - 1
7	010A		BNE	SUMA	es el contador = cero ?
8	010C		STAA	$40	guarde la suma
9	010E		SWI		interrupción por software

1. Resultados del análisis lexicográfico

Analicemos una parte del programa presentado:

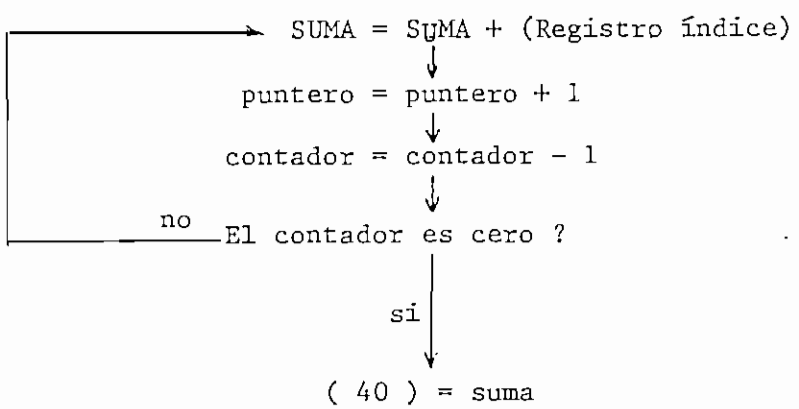
Los tokens en que se dividen las instrucciones, sus valores y clases se muestran en la tabla 3.1.

TABLA 3.1

# instrucción analizada	tokens	Parte de clase	Parte de valor
4.	SUMA	etiqueta	puntero a archivo de etiquetas
4.	ADDA	operador	puntero a archivo códigos nmotécnicos
4.	X	operando	registro índice
5.	INX	operador	puntero a archivo códigos nmotécnicos
6.	DECB	operador	puntero a archivo códigos nmotécnicos
7.	BNE	operador	puntero a archivo códigos nmotécnicos
7.	SUMA	etiqueta	puntero a archivo de etiquetas
8.	STAA	operador	puntero a archivo códigos nmotécnicos
8.	$40	constante hexadecimal	valor 40 hexadecimal

2. Resultados del análisis sintáctico-semántico

Arbol sintáctico (de instrucción 4 a instrucción 8)



STAA	$40	tiene direccionamiento directo
BNE	SUMA	tiene direccionamiento relativo
ADDA	X	tiene direccionamiento indexado

Todos los identificadores son válidos.

3. Tabla de Simbolos (archivo de etiquetas)

(de instrucción 4 a instrucción 8)

SUMA - 0106

4. Tabla de símbolos.

(códigos nmatécnicos usados y sus hexadecimales correspondientes) (de instrucción 4 a instrucción 8)

	D1	D2	D3	D4	D5	D6
ADDA	8B	9B	AB	BB		
INX					08	
DCB					5A	
BNE						26
STAA		97	A7	B7		

D1 - direccionamiento inmediato

D2 - direccionamiento directo

D3 - direccionamiento indexado

D4 - direccionamiento extendido

D5 - direccionamiento implícito

D6 - direccionamiento relativo

5. Generación de código

(de instrucción 4 a instrucción 8)

# instrucción	dirección memoria	contenidos de memoria (hexadecimal)
4.	0106	AB 00
5.	0108	08
6.	0109	5A
7.	010A	26 FA
8.	010C	97 40

3.3.6 Manejo de errores.

En un assembler, generalmente, la manipulación de errores se limita a la detección y reporte de los mismos. Algunos ensambladores más complejos realizan, además, una modificación de la entrada incorrecta de forma que se pueda seguir compilando el programa y detectar tantos errores como sea posible.

Sin embargo, en la mayoría de los casos el ensamblador detiene el proceso después de la detección del primer error. Este debe ser corregido por el usuario y nuevamente ingresado para ser compilado. De esta manera se procede hasta que el programa se depura totalmente y el código objeto es generado.

En un cross-assembler que se corre usualmente en una computadora grande que dispone de terminales iterativos, este proceso se efectúa rapidamente, no representa una molestia significativa y en cambio constituye una ventaja en cuanto a costos en comparación a ensambladores con tratamiento de errores con mecanismos de separación o corrección.

En las distintas etapas del proceso de ensamblaje se detectan errores y la subrutina que los maneja envía mensajes de diagnóstico de los mismos, en los cuales explica en qué consisten. Estos mensajes deben tener las mismas características de claridad y no redundancia que se especificaron en el Capítulo II para compiladores de lenguajes más sofisticados.

3.3.7 Tablas de símbolos.

Normalmente la información de las tablas se va llenan-

do de una de las dos formas siguientes:

1. El valor de las etiquetas y sus nombres se captan en una primera leída. El resto de información necesaria se obtiene en las distintas etapas de análisis.

2. La información se va recolectando y colocando en las tablas, según se van analizando secuencialmente las instrucciones.

En el primer caso el assembler es de más de una pasada; en el segundo, se tiene un ensamblador de una pasada.

Los programas en assembler no son muy largos y las tablas de símbolos no tienen un gran número de entradas. Por esta razón, a menudo sobre todo en ordenadores grandes y rápidos, las búsquedas se realizan en una tabla sin clasificar con un promedio para una lista de n componentes de (N + 1)/2 búsquedas.

C A P I T U L O IV

APLICACION DE LA TEORIA DE DISEÑO DE COMPILADORES AL DISEÑO DEL COMPILADOR PROPUESTO, IMPLEMENTACION DE LOS PROGRAMAS DE COMPUTADORA EN FORTRAN IV, DESCRIPCION DE LOS MISMOS.

4.1 CONFIGURACION DEL CROSS-ASSEMBLER DESARROLLADO

Como se vió en el Capítulo anterior, en la programación de un assembler, ciertas partes de las tres fases de análisis: el buscador, el reconocedor y el análisis semántico, se encuentran a menudo en un mismo programado, debido a que son análisis no muy complejos y que además están íntimamente relacionados entre sí.

El cross-assembler que se ha realizado, es de un paso y de dos pasadas.

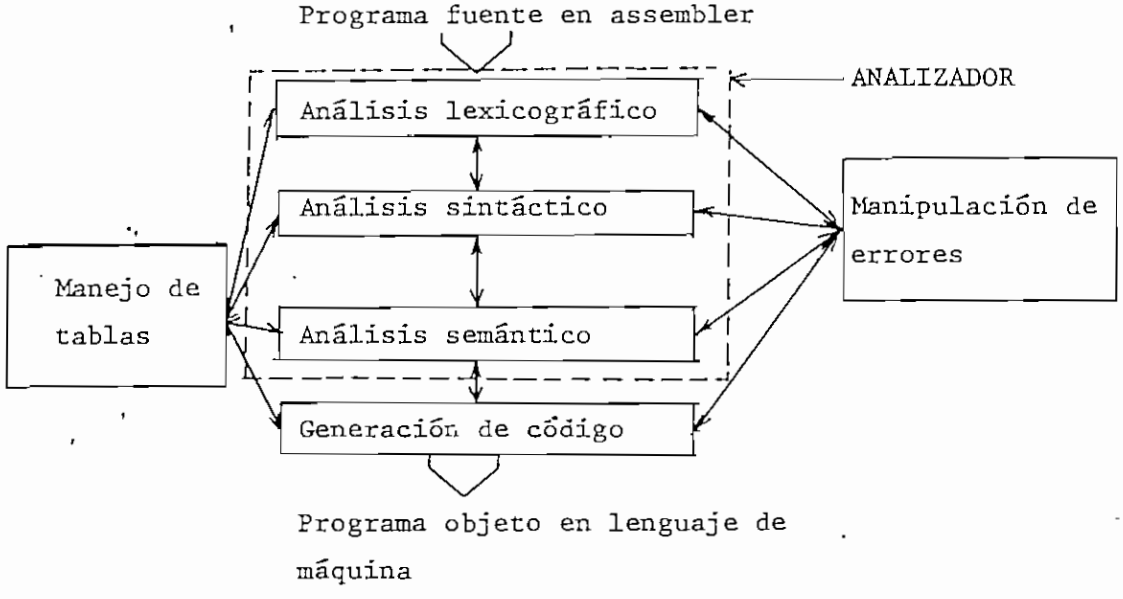
Por ser un paso y de acuerdo a lo que se vió anteriormente, todas las fases se hallan combinadas. Como todo el proceso se hace en un paso, se tiene que la operación de las fases puede actuar alternativamente con cambios de control entre fases diferentes. Debido a ser un ensamblador de un solo paso, no ocupa más

espacio en memoria que si tuviera múltiples pasos.

En un assembler de dos pasadas, primero se realiza una lectura del programa fuente, en la que se recolectan todos los símbolos y definiciones que aparecen y que serán útiles en la segunda pasada en el análisis, generación de código y detección de errores.

A continuación se verá el diagrama de bloques del cross-compilador desarrollados.

Fig. 4.1 Diagrama de bloques del cross-assembler desarrollado para la M6800.



Cada fase realiza las funciones tal como se explicaron en el Capítulo anterior.

Las tablas son no ordenadas y con búsqueda secuencial.

Los errores se detectan, los mensajes correspondientes se imprimen. No se detiene la compilación al encontrar un error, sino que puede seguirse ensamblando el programa y detectando entradas incorrectas.

En el Apéndice C se encuentran las tablas de mensajes de error. En el Apéndice B se encuentran los listados de los programas con las siguientes opciones. (Los programas más importantes del sistema cross-assembler, tienen listados y mapas resultantes de la compilación, el resto de programas sólo tienen listados).

XREF : listado del contenido de la tabla de símbolos

SYMBOLS: listado del programa fuente

A partir del listado obtenido utilizando estas opciones, se pueden observar entre otras cosas, lo siguiente:

- Las localizaciones de variables, su equivalencia y variables internas.
- La localización de líneas de sentencia en memoria
- Los periféricos utilizados por el computador
- Los subprogramas llamados

- una lista de todos los símbolos y constantes de la tabla respectiva con su utilización, tipos de longitudes y referencias.

- Tipos de sentencias usadas y referencias.

PROGRAMAS QUE CONFORMAN EL ASSEMBLER. DESCRIPCION DE SUS FUNCIONES. TAREAS DEL PROCESO DE COMPILACION QUE EJECUTAN CADA UNO. DIAGRAMAS DE FLUJO.

Todos los programas están realizados en el FORTRAN del minicomputador DM-250 de la General Automation. Las variables son de precisión extendida y los enteros de dos palabras.

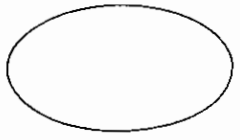
En el computador usado, los enteros de 2 palabras (1 palabra = 16 bits = 2 bytes), tienen un máximo valor posible de 2147483647.

Precisión extendida produce que todas las constantes reales ocupen tres palabras de almacenamiento en memoria (6 bytes = 48 bites).

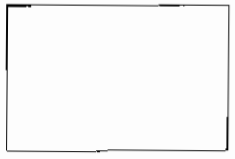
Los símbolos usados en los diagramas de flujo, son los mostrados en la Fig. 4.1

Tab. 4.1

Símbolos standard para diagramas de flujo.



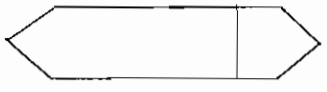
Punto terminal de inicio o final



Procesamiento aritmético y movimiento de datos.

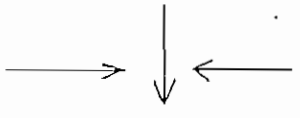
Decisión

Llamada a subrutina

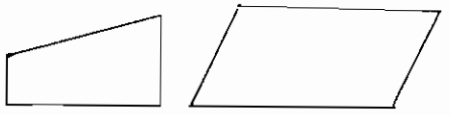


Operación "DO"

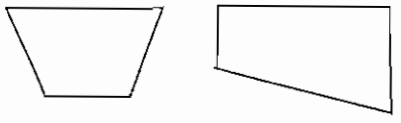
Punto de conexión



Flecha de conexión



Lectura , lectura de disco



Escritura , escritura en disco

4.2 A R C H I V O S

Definición : DEFINE FILE A (X, Y, U, J)

A - número del archivo

X - número de registros contenidos en el archivo

Y - número de palabras por registro

U - archivo sin formato

J - puntero del archivo

ARCHIVO 1 : NMONI

Localización : ALLOCATE NMONI, S (401B) SECTORES 40

Definición : DEFINE FILE 1 (400, 32, U, J1)

Propósito : Archivo en el que se guardan los códigos nmotécnicos y sus respectivos códigos hexadecimales.

Nomenclatura : NMON - Código nmotécnico

D1 - direccionamiento inmediato

D2 - direccionamiento directo

D3 - direccionamiento indexado

D4 - direccionamiento extendido

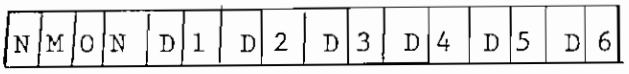
D5 - direccionamiento implícito

D6 - direccionamiento relativo

Diseño del archivo: Cada registro del archivo contiene un código nmotécnico y sus códigos hexadecimales de acuerdo al tipo de direccionamiento.

La estructura de los registros es la siguiente:

Fig. 4.2



Tiene 16 elementos enteros de 2 palabras cada uno, es decir está constituído por 16 palabras organizadas así:

4 elementos - 8 palabras para el código nmotécnico

2 elementos - 4 palabras para el código hexadecimal correspondientes a cada tipo de direccionamiento.

NOTA : 1 Sector = 320 palabras
En este archivo un registro = 32 palabras
Todo el archivo tiene 32 x 400 = 12800 palabras: 40 sectores.

ARCHIVO 2 : W : SFL

Localización : ALLOCATE SFL.S (401B)SECTORS 20

Definición : DEFINE FILE 2 (500, 12, U, J2)

Propósito : Archivo para almacenar etiquetas y demás referencias de ese tipo (Tabla de símbolos).

Diseño del archivo : Cada registro tiene 6 posiciones de 2 palabras cada una, es decir 12 palabras. Su estructura es la siguiente:

Fig.4 .3

DISEÑO ARCHIVO SFL

L	A	B	E	L	#

Con 20 sectores se está permitiendo el almacenaje de:

320 x 20 = 6400 palabras

Esto es aproximadamente 530 registros. Por seguridad el número de etiquetas y referencias se limitarán a 520, como máximo dentro de un programa.

El primer registro del archivo contiene el número total de registros, incluyendo el primero

ARCHIVO 3 : ASCII

Localización : ALLOCATE ASCII.S (4010) SECTORS 5

Definición : DEFINE FILE 3 (250, 6, U, J3)

Propósito : Archivo que guarda los códigos ASCII válidos para este assembler.

Diseño del archivo : Tiene 3 posiciones por registro: 6 palabras. Su configuración es la que sigue:

Fig. 4.4

DISEÑO ARCHIVO ASCII

NC	LCD(1)	LCD(2)

Nomenclatura : NC - carácter ASCII

LCD(1) - primer dígito de representación interna del carácter

LCD(2) - Segundo dígito de representación interna del carácter.

Con 5 sectores se pueden ingresar al archivo 1600, palabras aproximadamente 250 registros.

El primer registro del archivo guarda el número total de registros, incluyendo el primero.

ARCHIVO 4 : DEL PROGRAMA EN ASSEMBLER

Localización : ALLOCATE NOMBRE-PROGRAMA.S (401B) SECTORS X

$$X = \frac{\#\ \text{líneas de programa}}{2} + 1$$

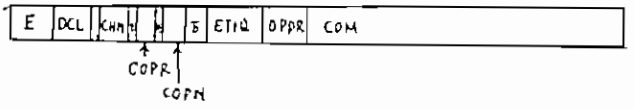
Definició n : DEFINE FILE 4 (1000, 160, U, J4)

Propósito : Archivo en el que se almacena uno o varios programas que se desean ensamblar. El número total de líneas de programa debe ser menor a 1000.

Diseño del archivo: Es un archivo de 80 posiciones, cuya estructura se muestra a continuación:

Fig. 4.5

DISEÑO ARCHIVO PROGRAMA FUENTE EN ASSEMBLER



donde:

E - error

DCL - contador de líneas (decimal)

CHN - Contador hexadecimal de memoria

COPR - Código hexadecimal del operador

COPN - Código hexadecimal del operando

ETIQ - Etiqueta

OPDR - Operador

OPDN - Operando

COM - Comentarios

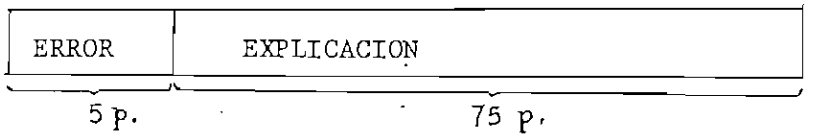
b - Blanco

El primer registro contiene el número total de registros, incluyendo al puntero.

ARCHIVO 5	: ALLOCATE NERRO.S' (401B) SECTORS 50
Definición	: DEFINE FILE 5 (100, 160, U, J5)
Propósito	: Archivo que contiene los códigos de error y los mensajes de explicación de los mismos.
Diseño del archivo	: Cada registro consta de 80 posiciones con un total de 160 palabras. Su diseño se muestra en la figura que sigue.

Fig. 4.6

DISEÑO DEL ARCHIVO NERRO



Con 50 sectores se pueden guardar 100 líneas de mensajes de error. Entre cada mensaje se debe dejar un espacio en blanco.

El primer registro del archivo corresponde al número de registros con información válida almacenados sin contar con el primer registro.

4.3 PROGRAMAS PRINCIPALES

PROGRAMA EDITOR

Propósito : Programa para dar facilidades de edición de un programa en assembler de la Motorola M6800. Este programa en assembler puede luego ser ensamblado, usando el ensamblador A6800 que forma parte de esta tesis.

Subprogramas llamados : ninguno

Forma de utilización : EDITOR

Diagrama de flujo : Fig. 4.7

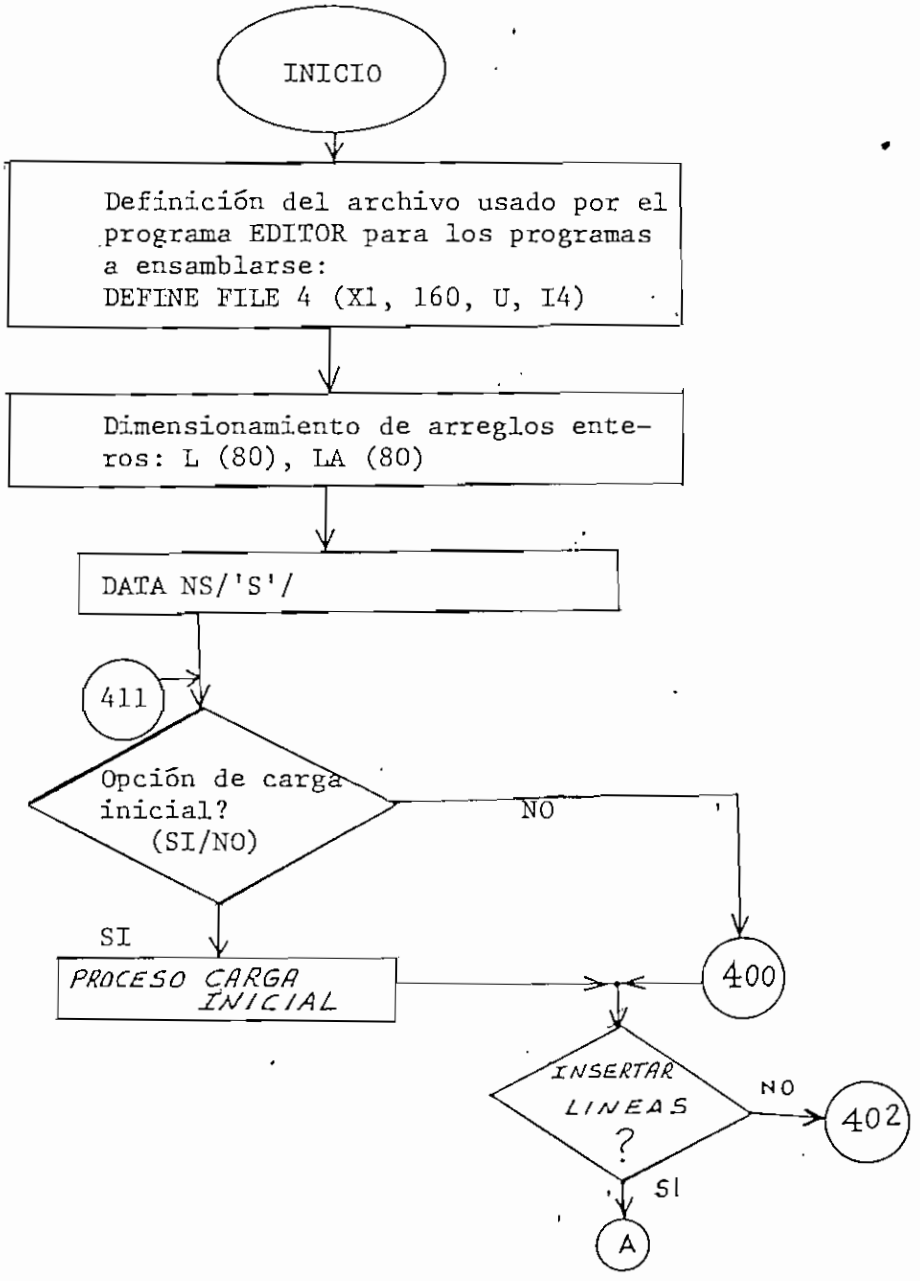
Listado : Ver Apéndice B, pág.236

Funciones que desempeña dentro del compilador:

No es directamente un bloque del compilador, pero tiene gran importancia ;pues facilita su funcionamiento,ya que , permite realizar la carga y corrección de programas en assembler en forma interactiva y versátil.

Fig. 4.7

PROGRAMA EDITOR



NOTA: X1, es por lo menos igual que el número de líneas del programa en assembler, aúnque para permitir correciones e inserciones, en mejor que se definan más líneas.

Fig. 4 .7 PROGRAMA EDITOR

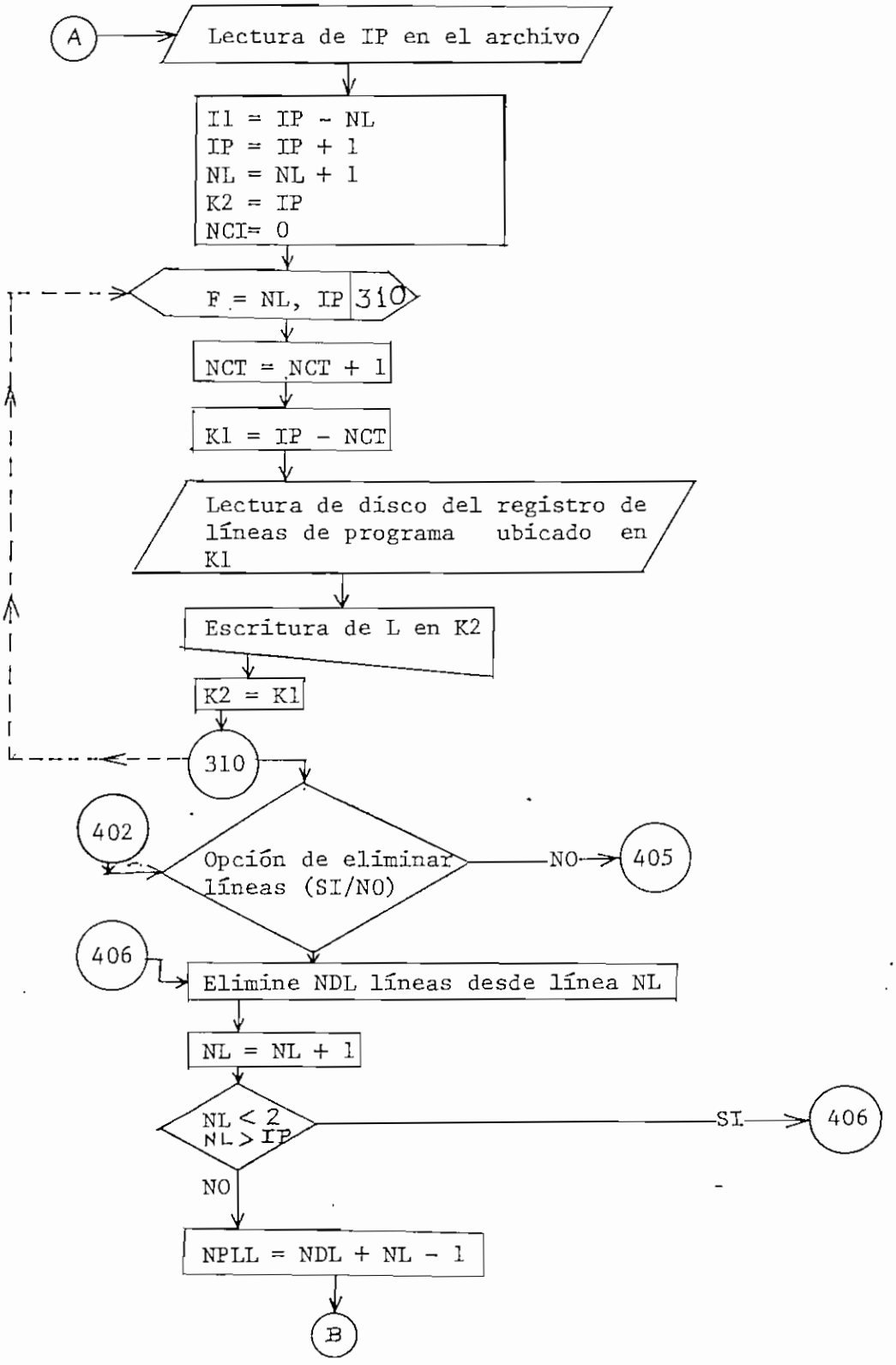


Fig. 4.7

PROGRAMA EDITOR

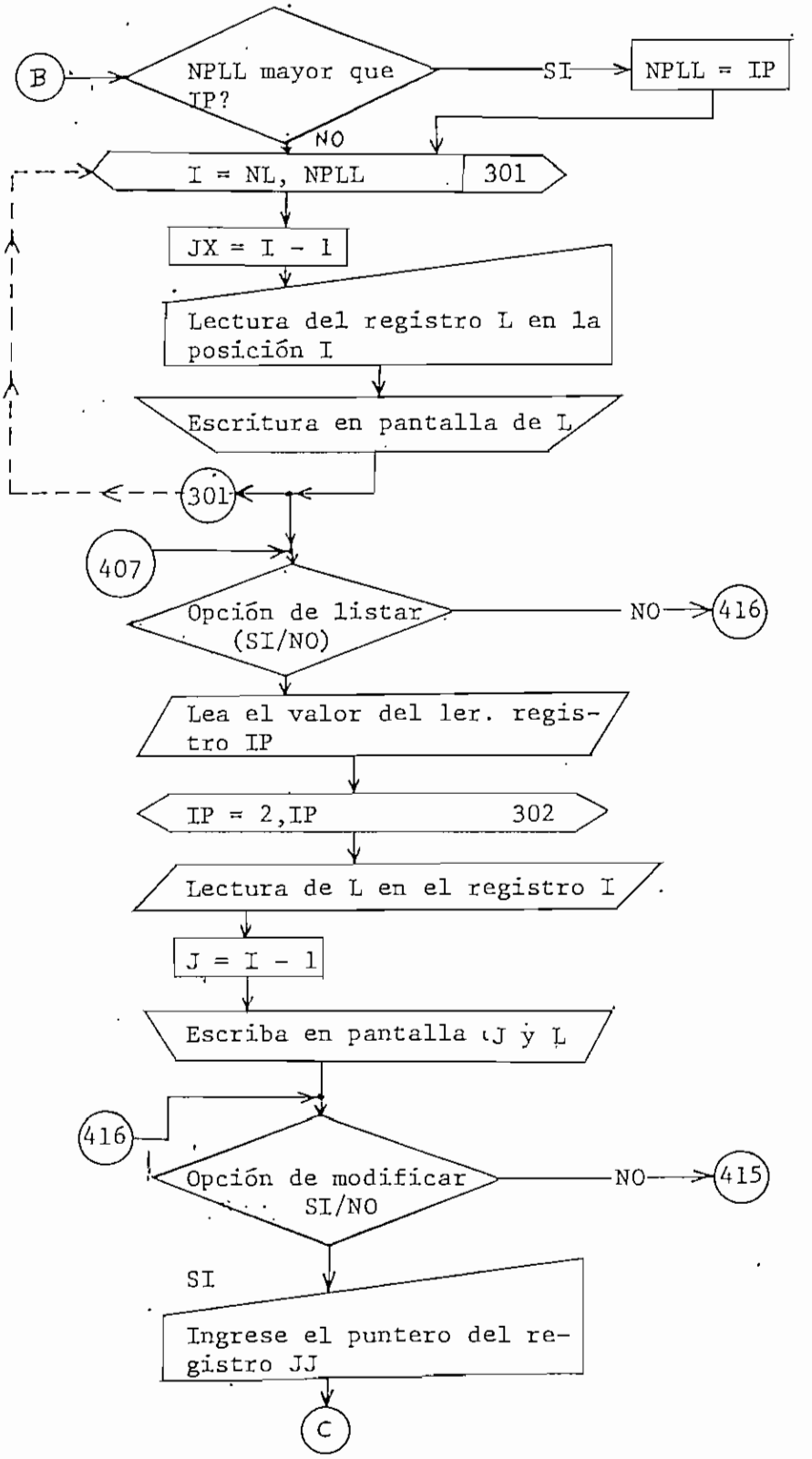
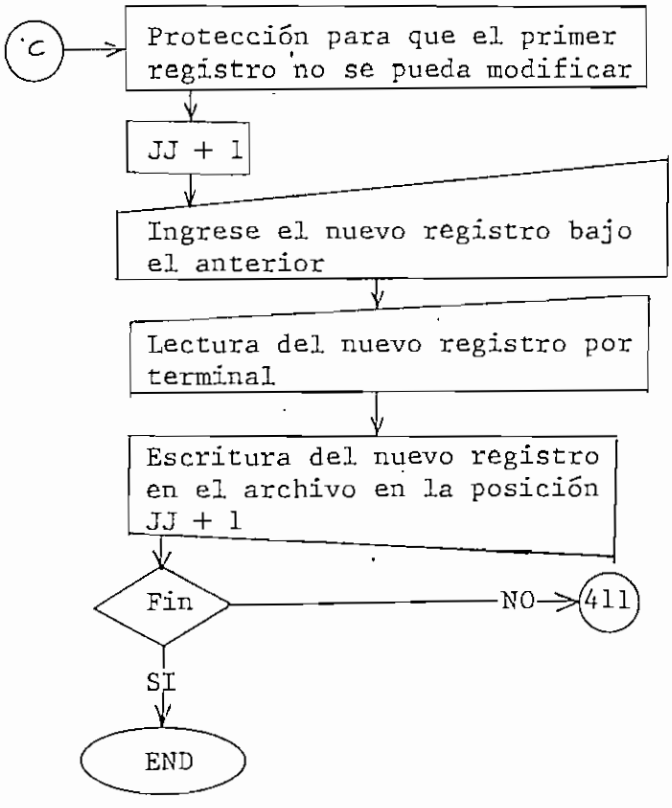


Fig. 4.7 PROGRAMA EDITOR



PROGRAMA A6800

Propósito	: Programa principal para el ensamblador de programas escritos en lenguaje assembler del microprocesador M6800.
Subprogramas llamados	: CGPR, VDF, PERR, SRORG, SREQU, SRRMB, SRFCC, SRFCB, CMP, VNMOP, MOVER,NTYPE, LABEL, INMED, RELAT, INDEX, DIREC, IMPLI, LCAR, COD1, COD24, COD3, COD5 COD6, DTH, NTACC.
Forma de utilización	: A6800
Diagrama de flujo	: Fig. 4.8
Listado	: Ver Apéndice B, pág. 232. Por la importancia que tiene este programa en el cross-assembler, se van a mostrar además de su listado, los mapas de referencias, símbolos y concordancias.

Funciones que desempeña dentro del compilador:

Este programa relaciona las distintas fases del assembler distribuídas en los subprogramas, las unifica y les hace funcionar en un solo paso. Organiza las pasadas del compilador e imprime ó muestra los resultados, es decir el programa en código objeto: contador de líneas de programa, contador hexadecimal de memoria, código hexadecimal del operando y equivalente hexadecimal del operando, si es que éste existe. Este programa ensamblador aparece a la izquierda del programa fuente en assembler.

Fig. 4.8

PROGRAMA A6800

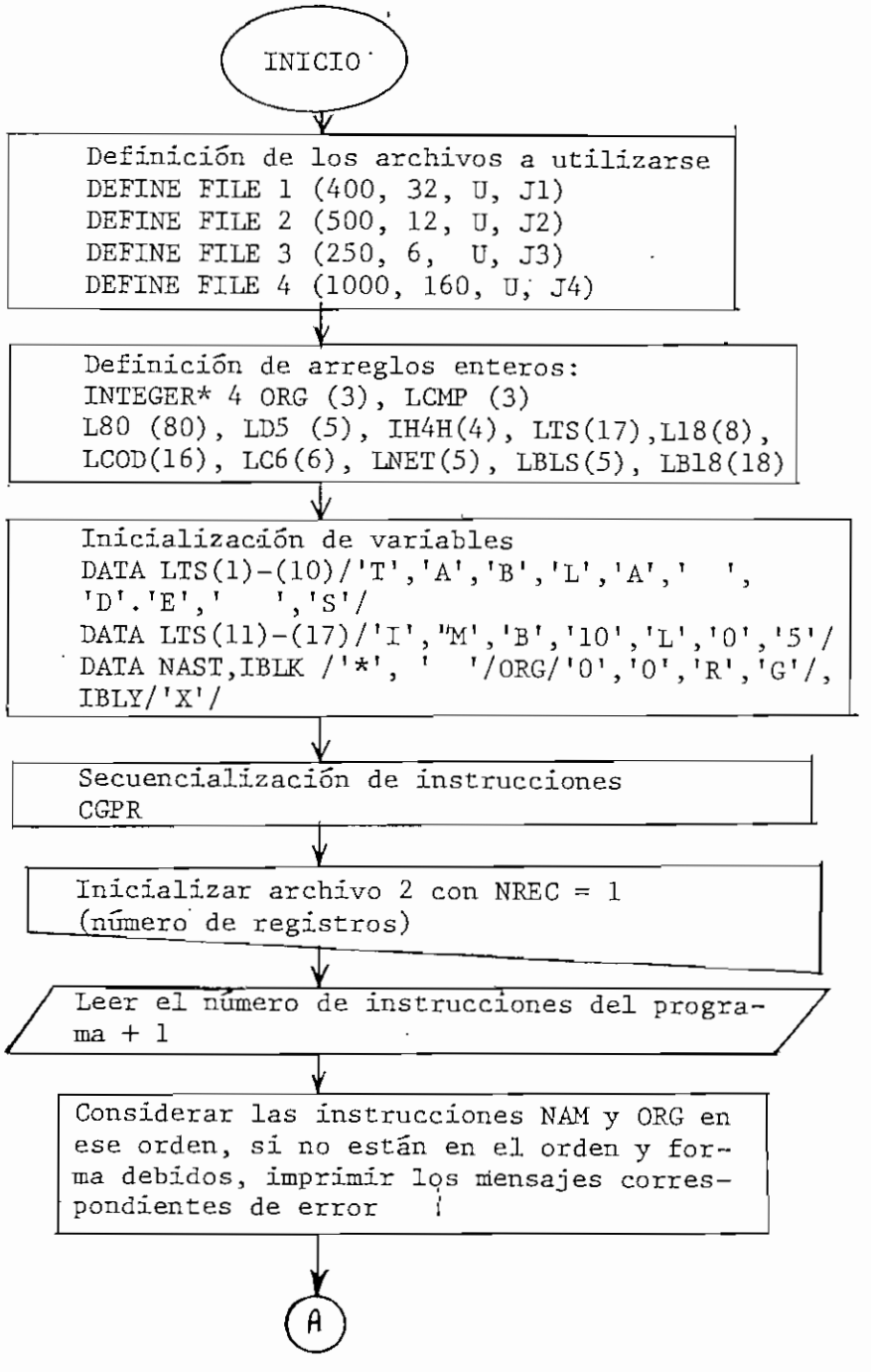


Fig. 4 . 8 PROGRAMA A6800

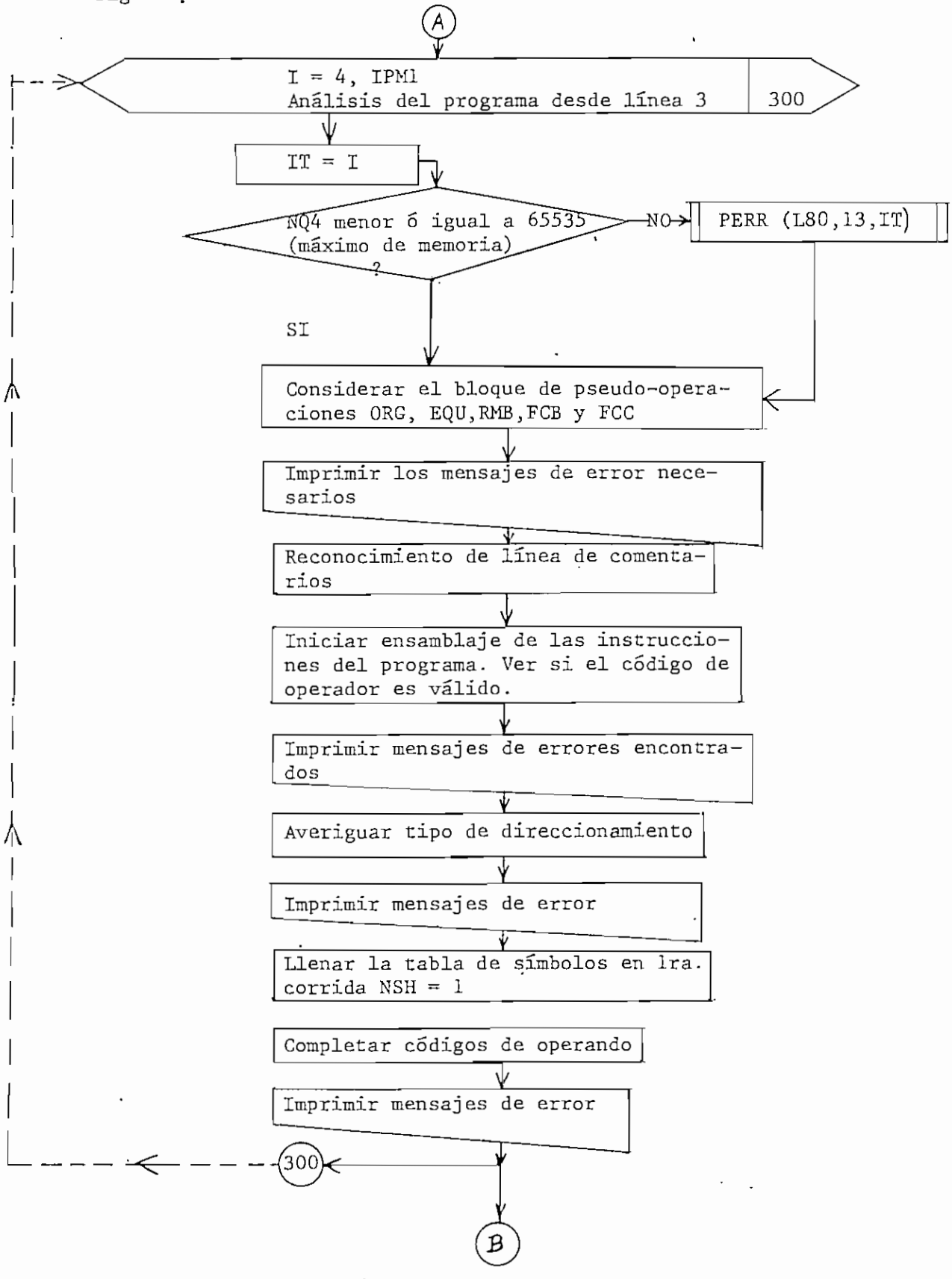
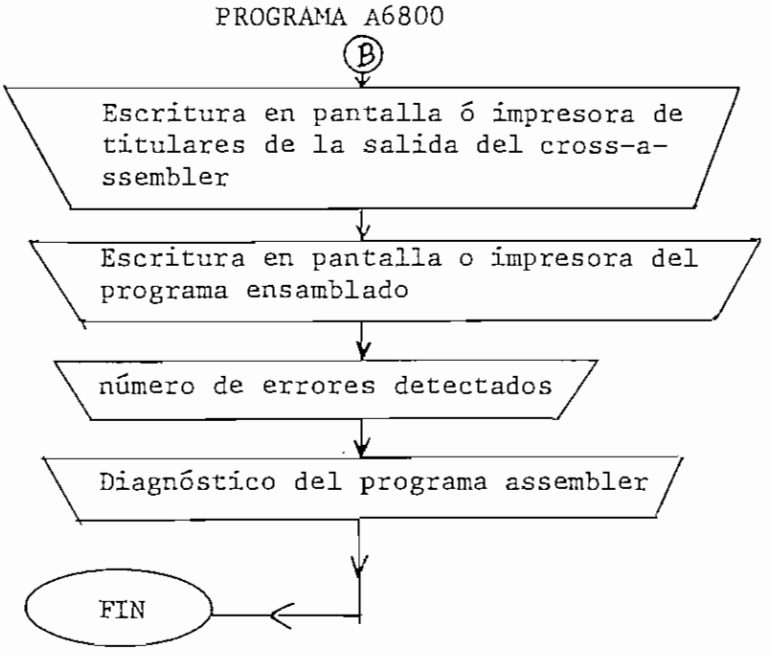


Fig. 4 .8


PROGRAMA A6800
B
Escritura en pantalla ó impresora de titulares de la salida del cross-assembler
Escritura en pantalla o impresora del programa ensamblado
número de errores detectados
Diagnóstico del programa assembler
FIN

PROGRAMA TANEM

Definición	: TANEM
Propósito	: Programa de carga del archivo NMONI
Subprogramas llamados	: Ninguno
Forma de utilización	: TANEM
Diagrama de flujo	: Fig. 4.9
Listado	: Ver Apéndice B, pág. 229

Funciones que realiza dentro del compilador:

Este programa sirve para cargar el archivo de los códigos nmatécnicos y todos los hexadecimales que le corresponden, de acuerdo a los diferentes tipos de direccionamientos.

Fig. 4.9

PROGRAMA TANEM

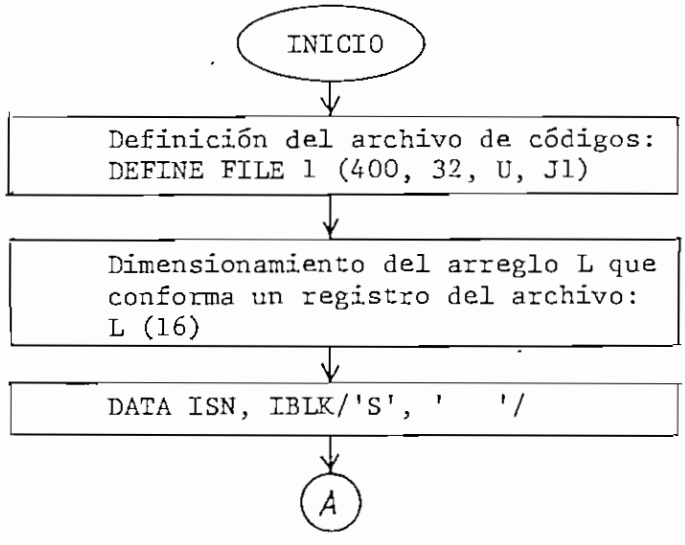


Fig. 4.9 PROGRAMA TANEM

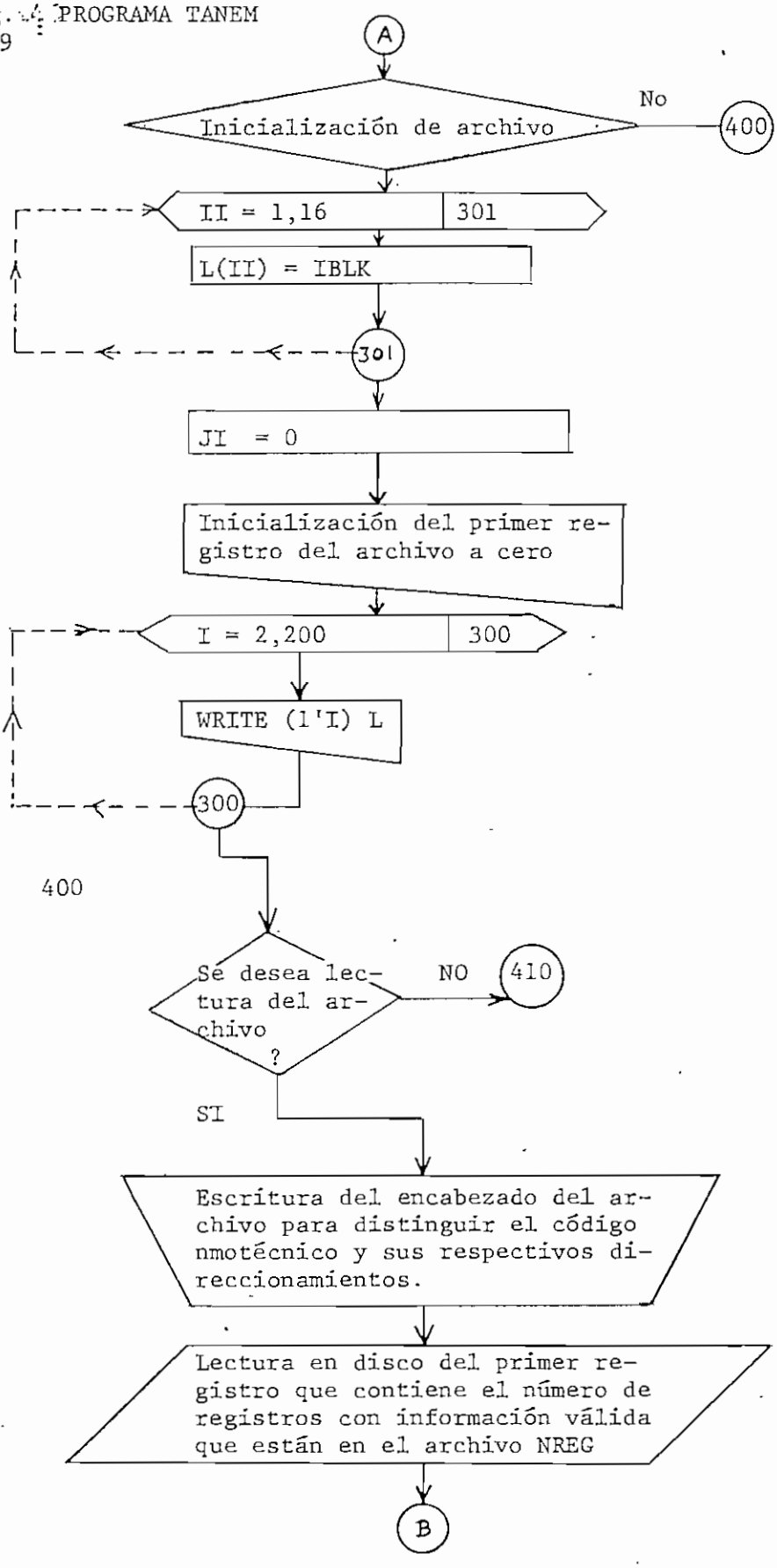


Fig. 4.9 : PROGRAMA TANEM

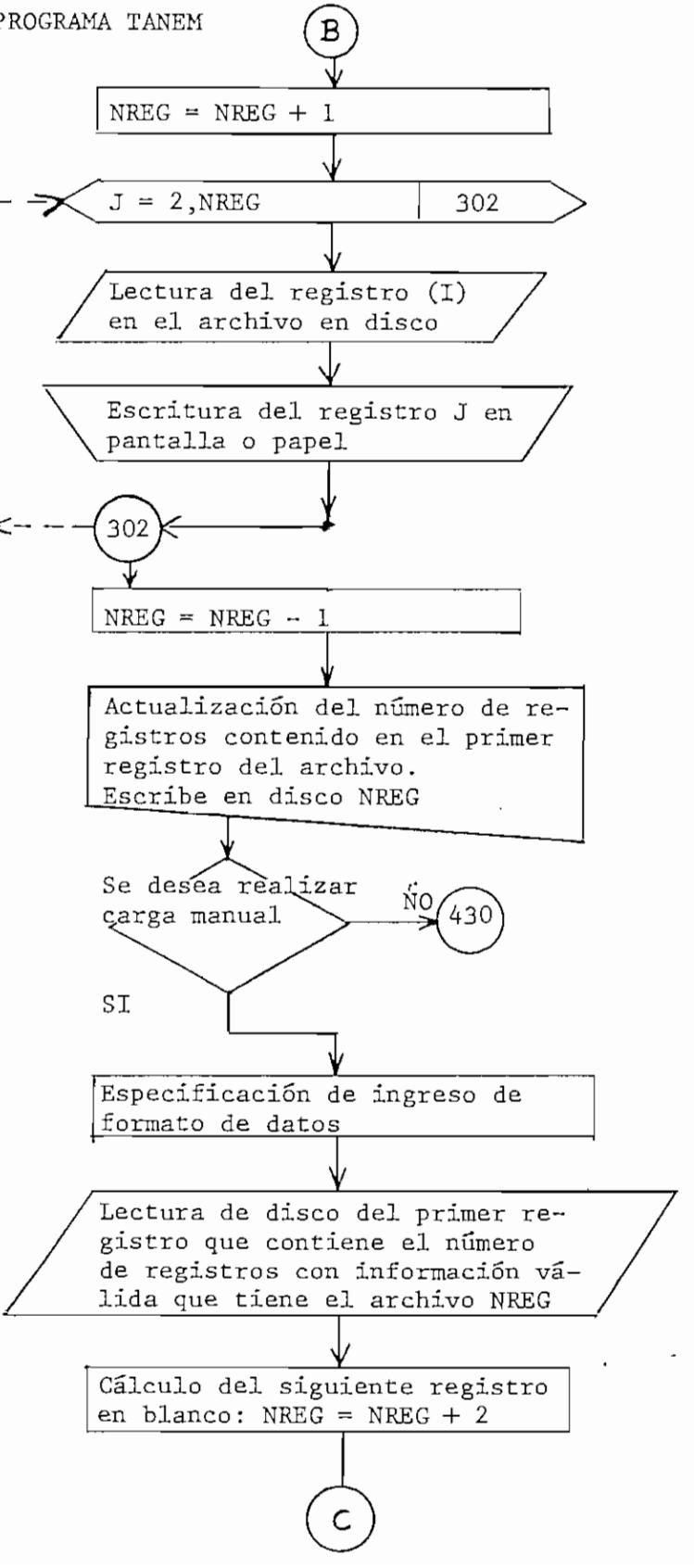


Fig. 4.9 PROGRAMA TANEM

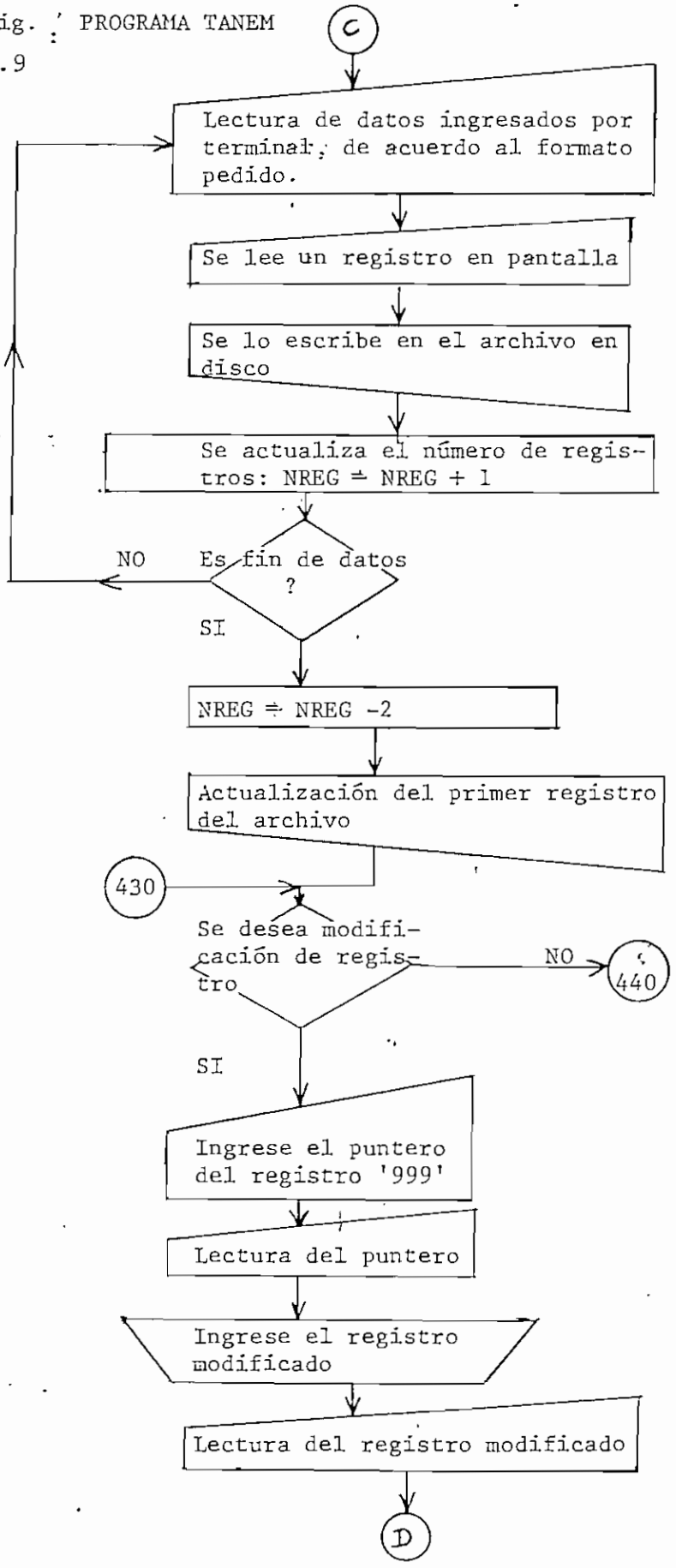
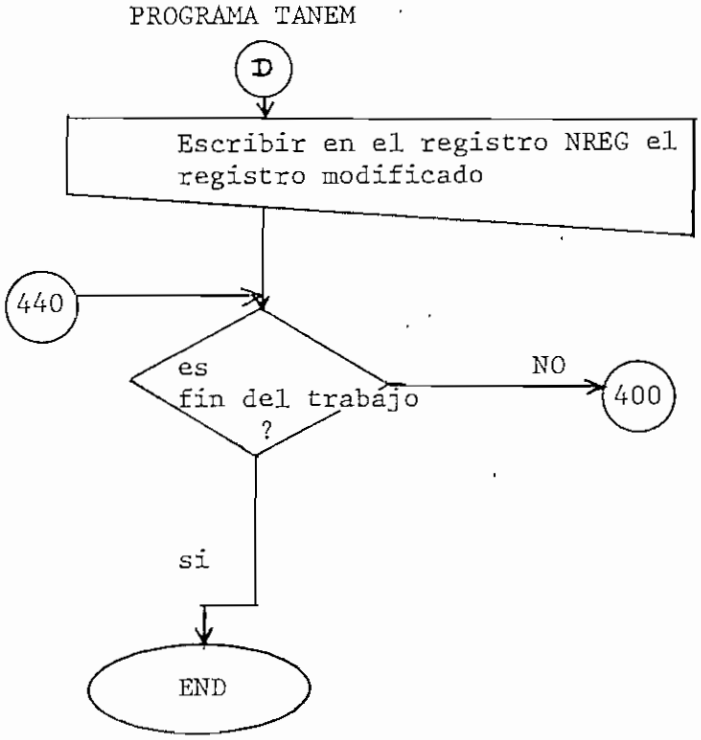


Fig. 4 .9


PROGRAMA TANEM
D
Escribir en el registro NREG el registro modificado
440
es fin del trabajo ?
NO
400
si
END

PROGRAMA LASCII

Propósito	: Programa para cargar la tabla con códigos ASCII
Subprogramas llamados	: ninguno
Forma de utilización	: LASCII
Diagrama de flujo	: Fig. 4.10
Listado	: Ver Apéndice B, pág.242

Funciones que desempeña dentro del compilador:

Es un programa de carga que no forma parte directamente del compilador, pero al crear la tabla de códigos ASCII, hace posible el funcionamiento del ensamblador con manejo de ese tipo de caracteres.

Fig. 4 .10

PROGRAMA LASCII

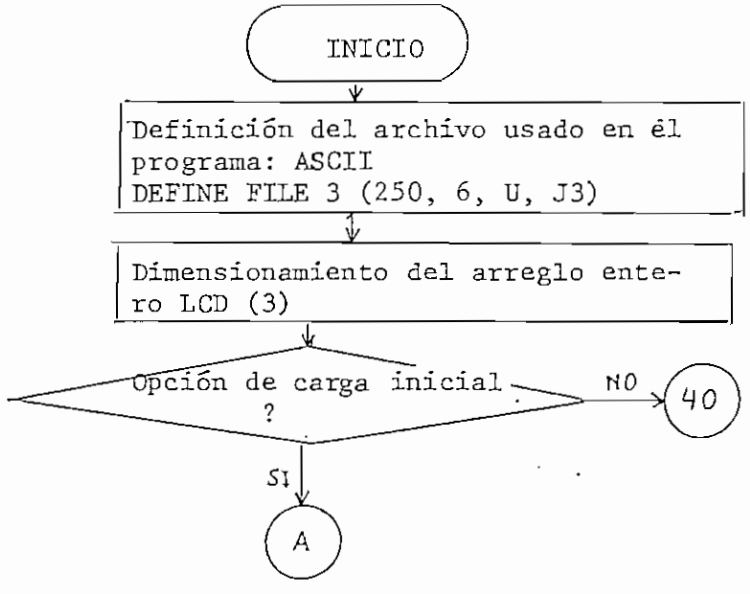
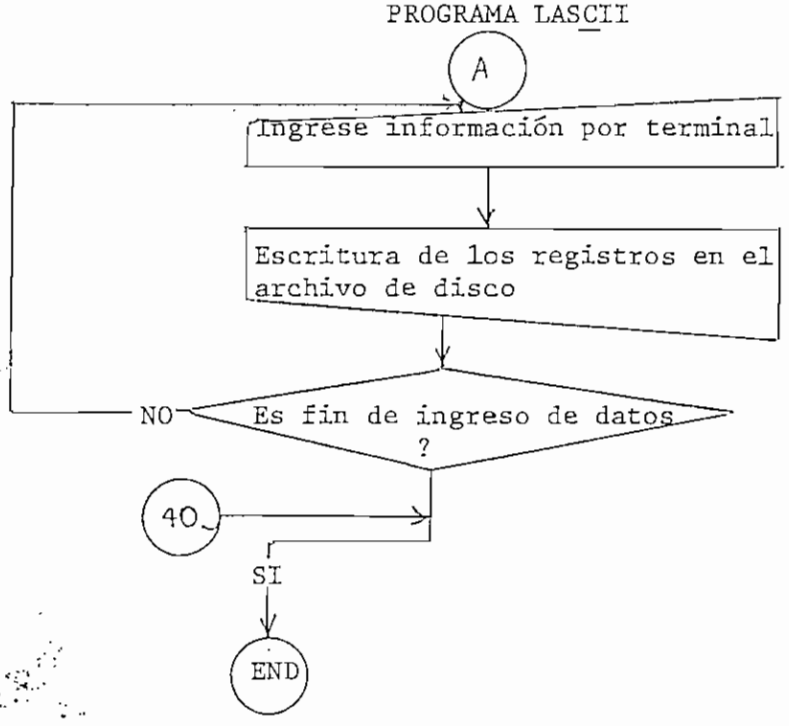


Fig. 4 .10


PROGRAMA LASCII
A
Ingrese información por terminal
Escritura de los registros en el archivo de disco
NO
Es fin de ingreso de datos ?
40
SI
END

PROGRAMA LERRO:

Propósito	: Programa de carga y lectura del archivo 5: NERRO. NERRO es el archivo con la descripción de los errores.
Subprogramas llamados	: Ninguno
Forma de utilización	: LERRO
Diagrama de flujo	: Fig. 4.11
Listado	: Ver Apéndice B, pág. 239

Funciones que desempeña:

Sirve para la carga y lectura del archivo de errores.

Fig. 4.11 PROGRAMA LERRO

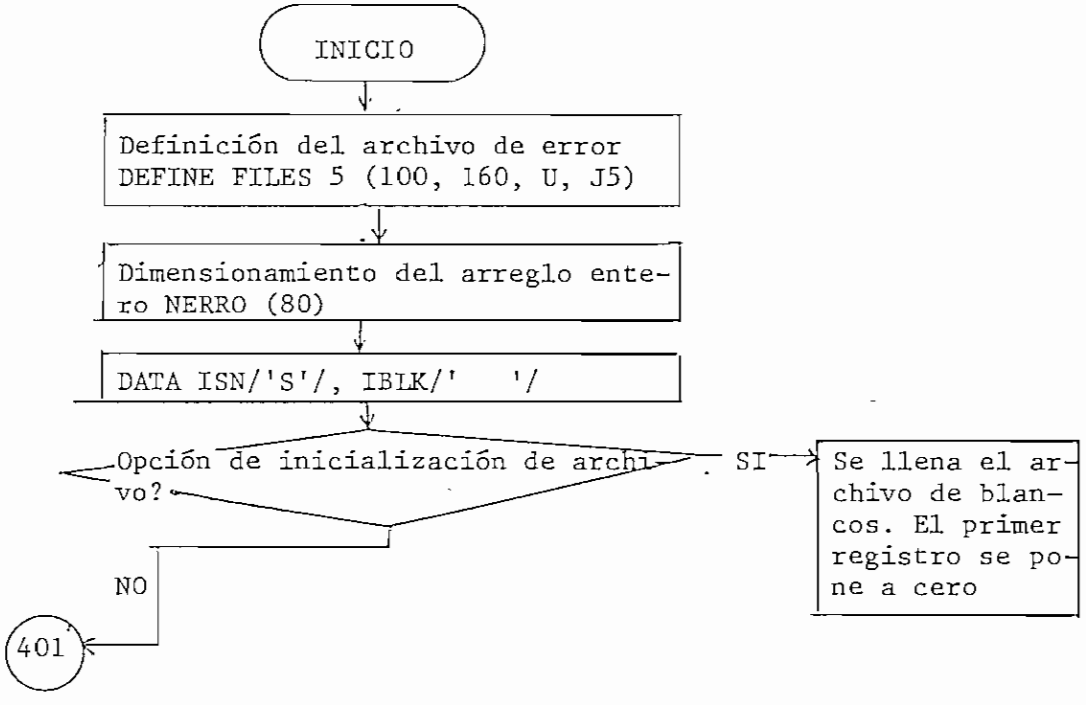
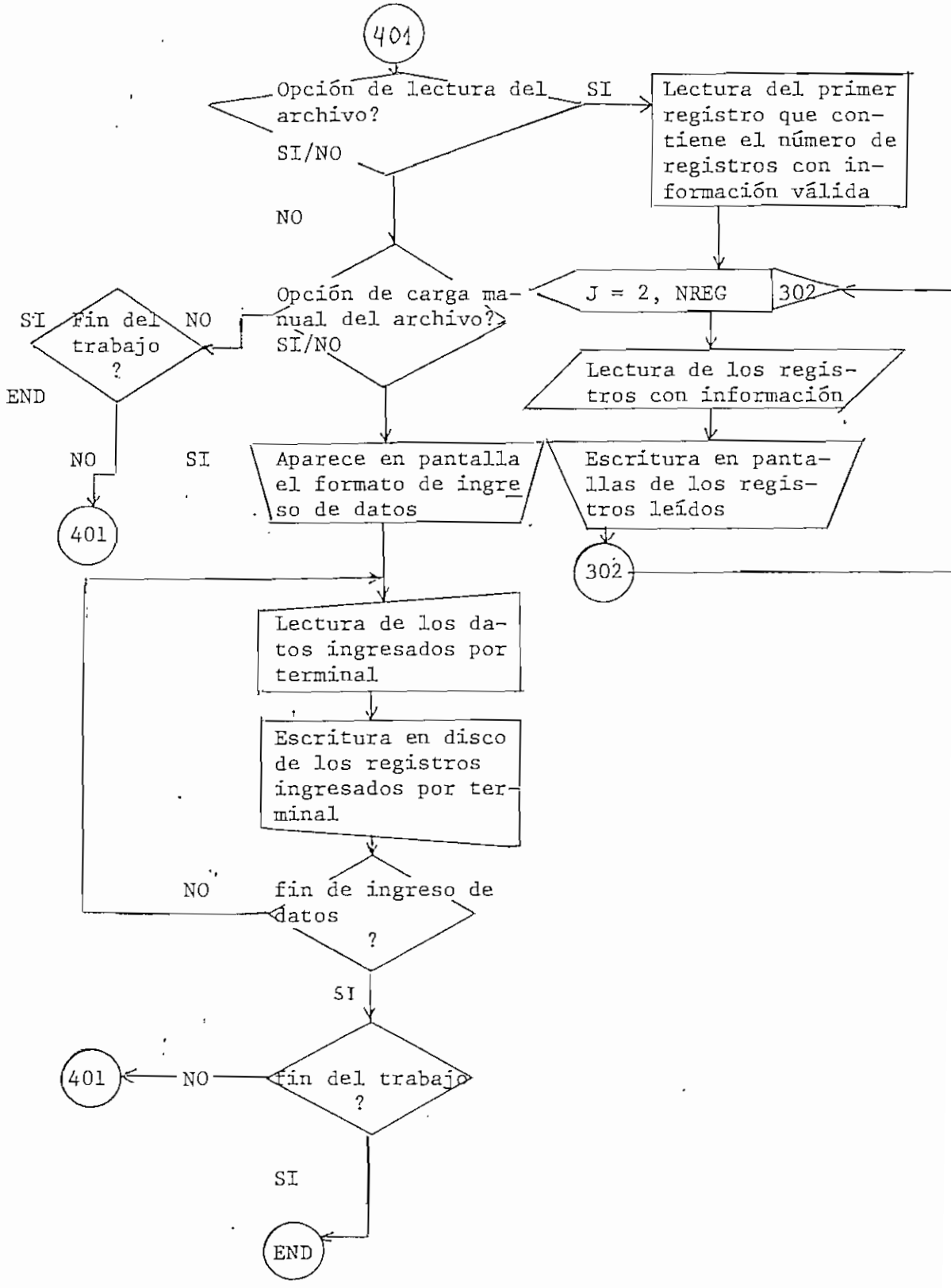


Fig. 4.11 PROGRAMA LERRO



PROGRAMA ERROR

Propósito	: Programa que a partir de un valor de error escribe en pantalla el significado de ese código de error.
Subprogramas llamados	: HRNTF, NTAAC
Forma de utilización	: ERROR
Diagrama de flujo	: Fig. 4.12
Listado	: Ver Apéndice B, pág.241

Funciones que desempeña dentro del compilador:

Es una importante parte del bloque de tratamiento de errores.

Fig. 4.12

PROGRAMA ERROR

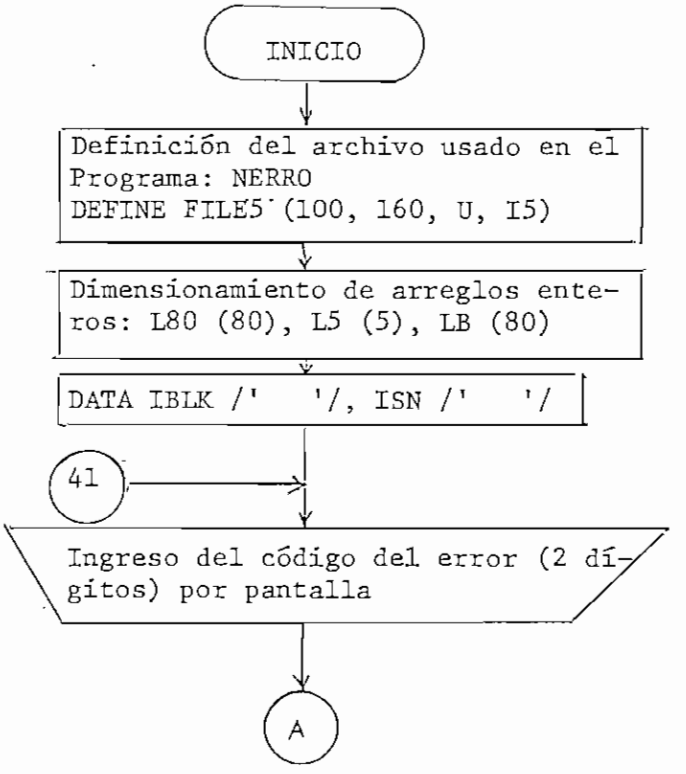
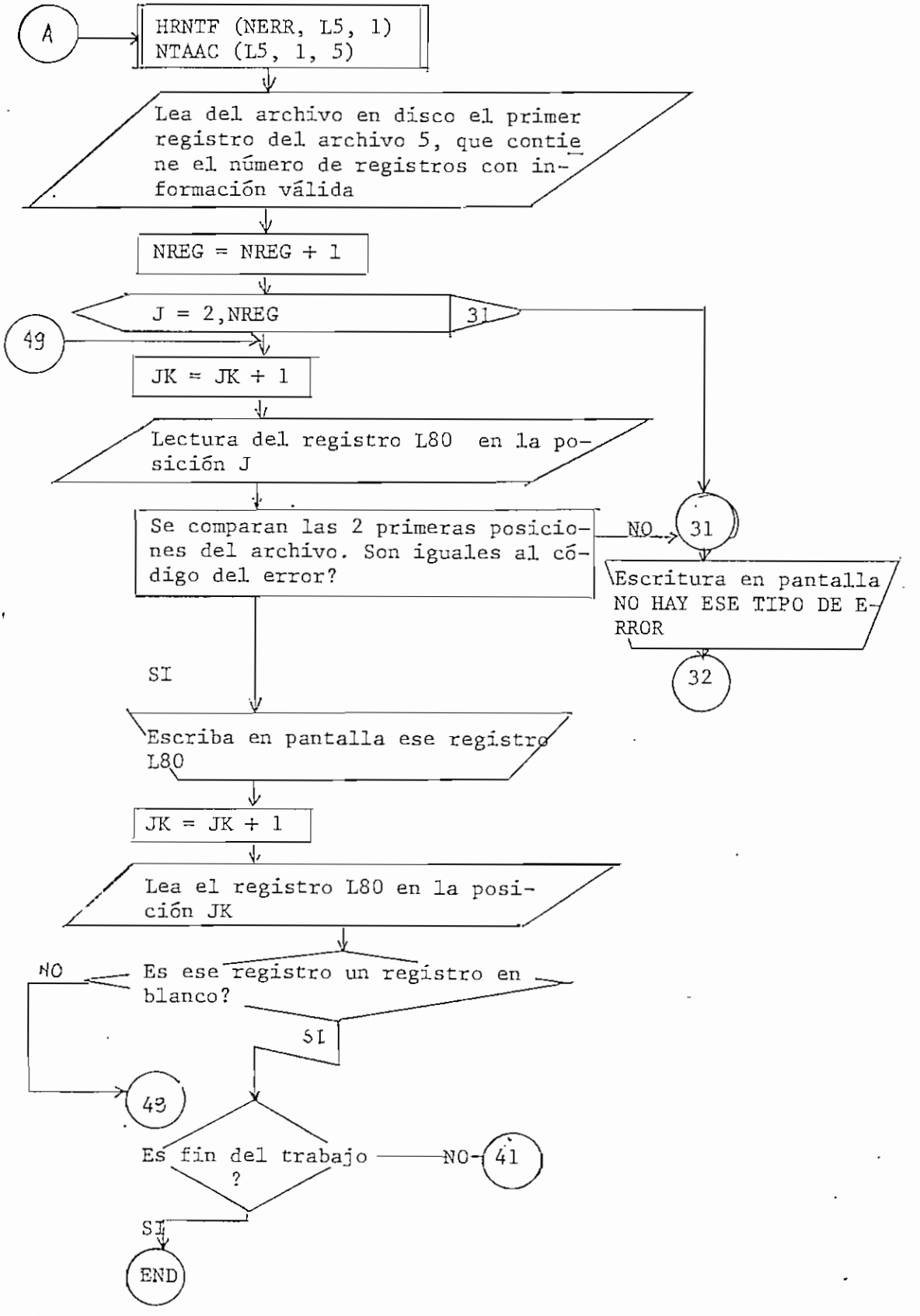


Fig. 4.12

PROGRAMA ERROR



PROGRAMA TEN

Propósito	: Programa para listar los símbolos cargados en el archivo ASCII.
Subprogramas llamados	: ninguno
Forma de utilización	: TEN
Diagrama de flujo	: Fig. 4.13
Listado	: Ver Apéndice B, pág.245

Funciones que realiza dentro del compilador:

No es parte integrante del compilador, pero realiza una tarea que tiene gran importancia: el mantenimiento de la tabla de códigos ASCII.

Fig. 4.13

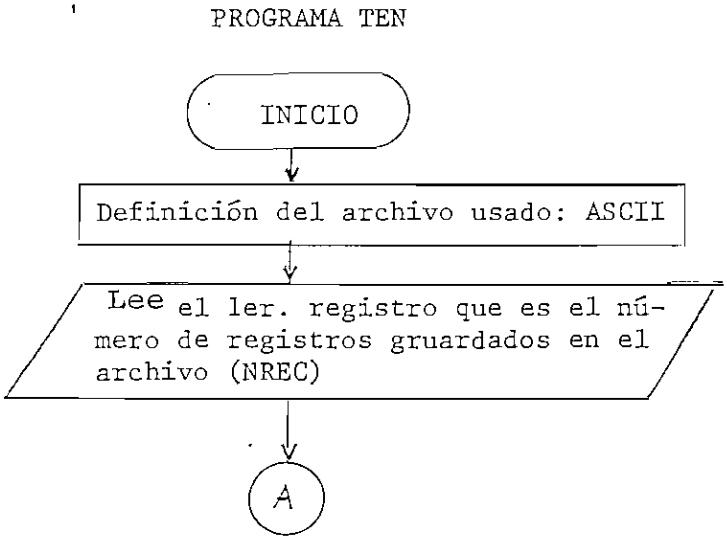
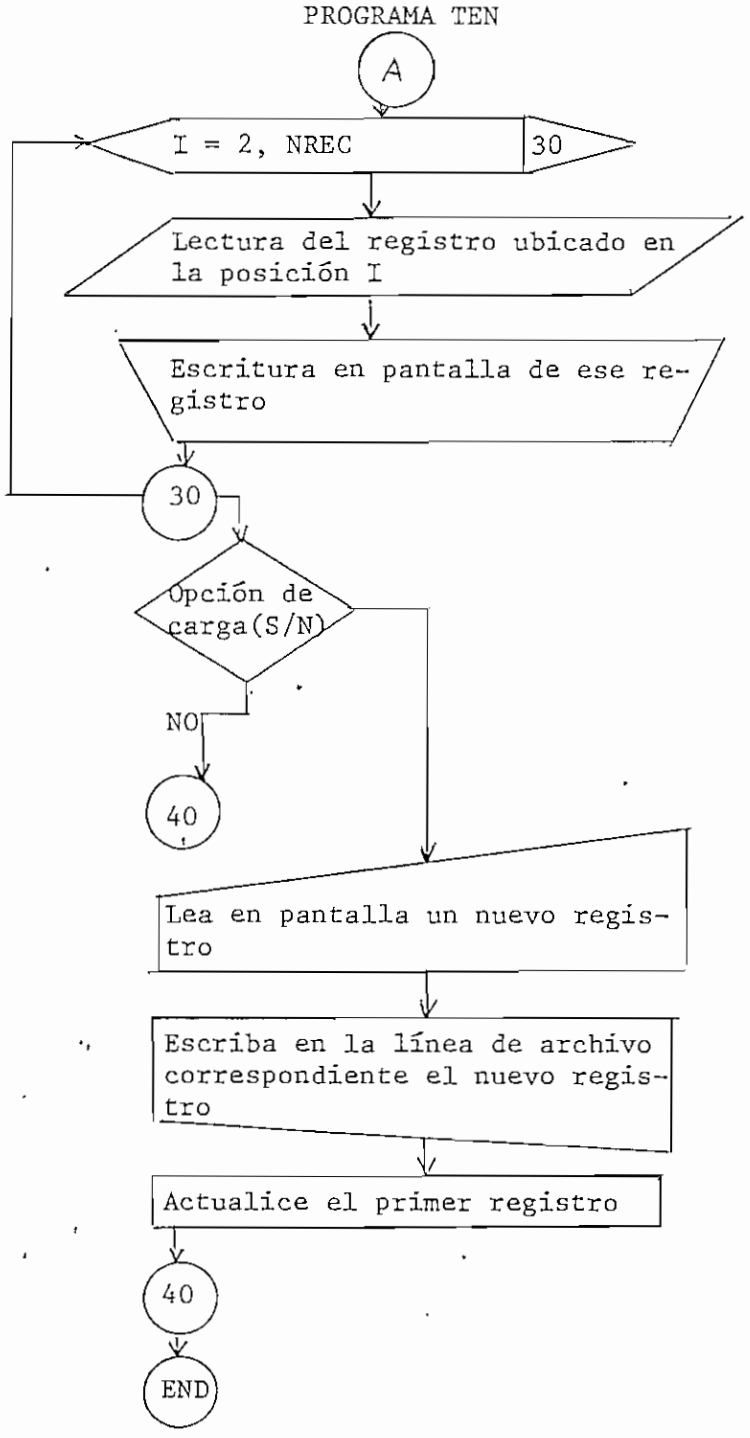


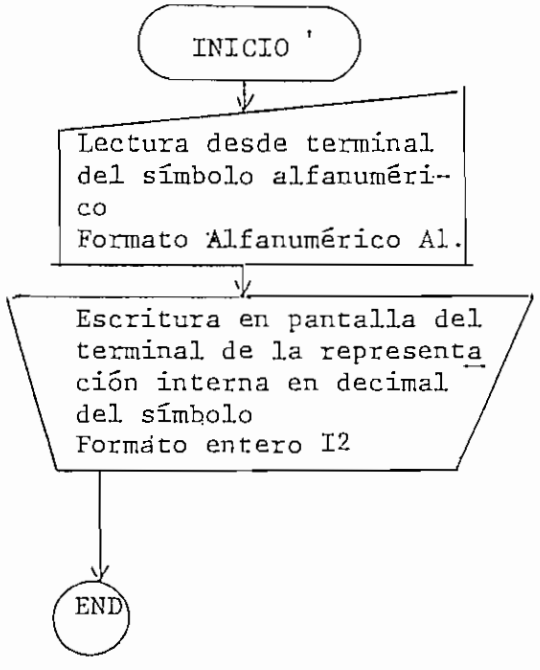
Fig. 4.13


PROGRAMA TEN
A
I = 2, NREC
30
Lectura del registro ubicado en la posición I
Escritura en pantalla de ese registro
30
Opción de carga(S/N)
NO
40
Lea en pantalla un nuevo registro
Escriba en la línea de archivo correspondiente el nuevo registro
Actualice el primer registro
40
END

PROGRAMA ALFCD

Definición	: ALFCD
Propósito	: Programa que entrega el número decimal correspondiente a un símbolo alfanumérico.
Subprogramas llamados	: Ninguno
Datos de entrada	: Se ingresa por pantalla de terminal el símbolo cuyo equivalente decimal se desea conocer.
Datos de salida	: El computador entrega el número decimal correspondiente, el mismo que aparece en pantalla.
Diagrama de flujo	: Fig. 4.14
Listado	: Ver apéndice B, pág. 246.

Fig. 4.14 PROGRAMA ALFCD



4.4 SUBRUTINAS LLAMADAS POR LOS PROGRAMAS DESCRITOS EN EL PUNTO 4.3

SUBRUTINA LABEL

Definición : SUBROUTINE LABEL (LA, NE, NDEC, LNET)

Propósito : Reconocimiento de etiquetas válidas

Subprogramas llamados : ATNAC

Forma de utilización : CALL LABEL (LA, NE, NDEC, LNET)

Parámetros : LA, NE, NDEC , LNET

LA: arreglo de 18 elementos, donde el subprograma recibe el operando

NE: 0 - etiqueta no válida

1 - etiqueta sola

2 - etiqueta + número válido

3 - etiqueta - número válido

4 - (etiqueta + número no válido) mayor que 65535

NDEC: equivalente decimal del número sumado a la etiqueta.

Datos de entrada : LA

Datos de salida : NE, NDEC, LNET

Diagrama de flujo : Ver Fig. 4.15

Listado : Ver Apéndice B, pág. 261.

Fig. 4·15 SUBRUTINA LABEL

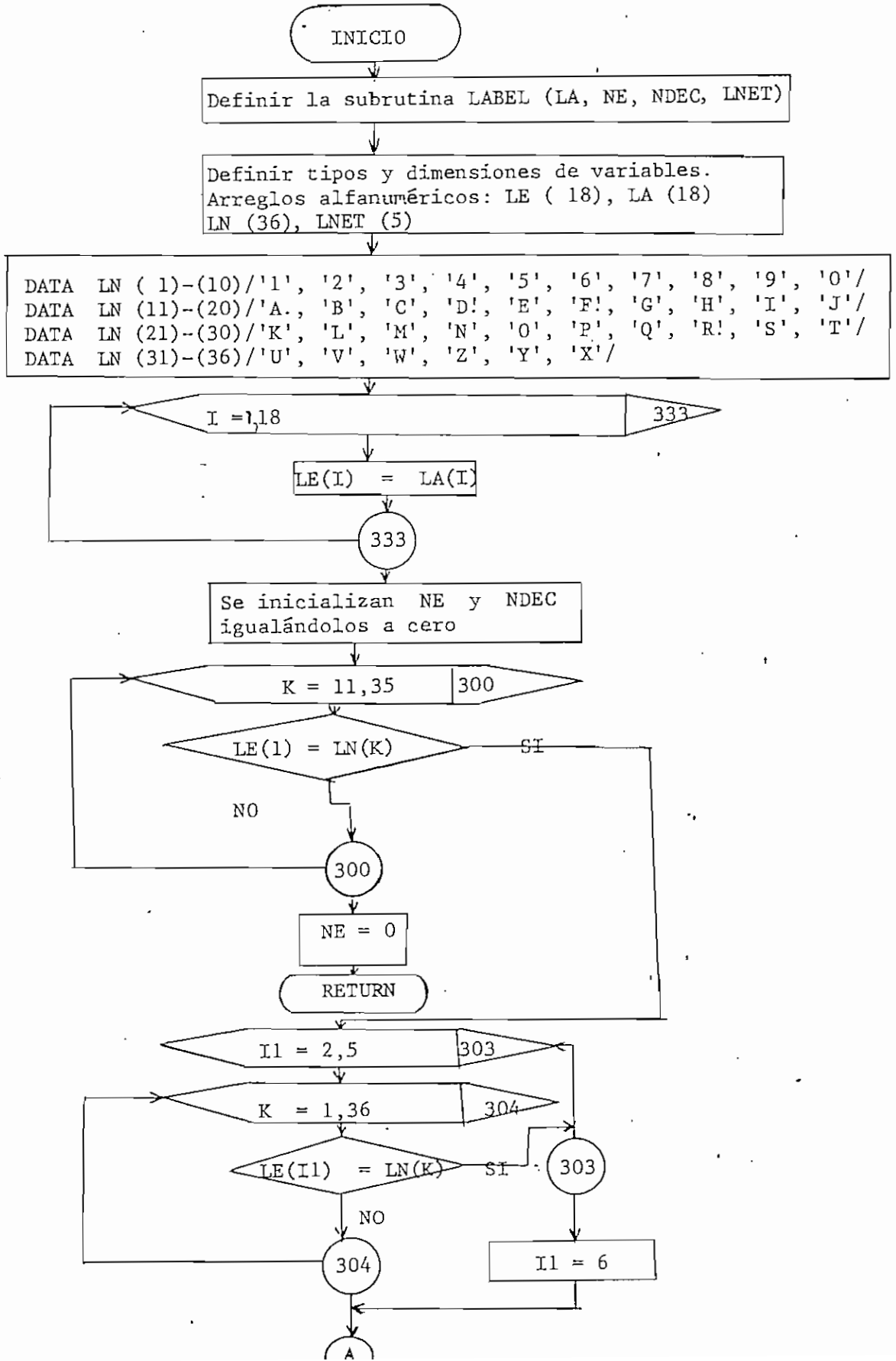


Fig. 4.15

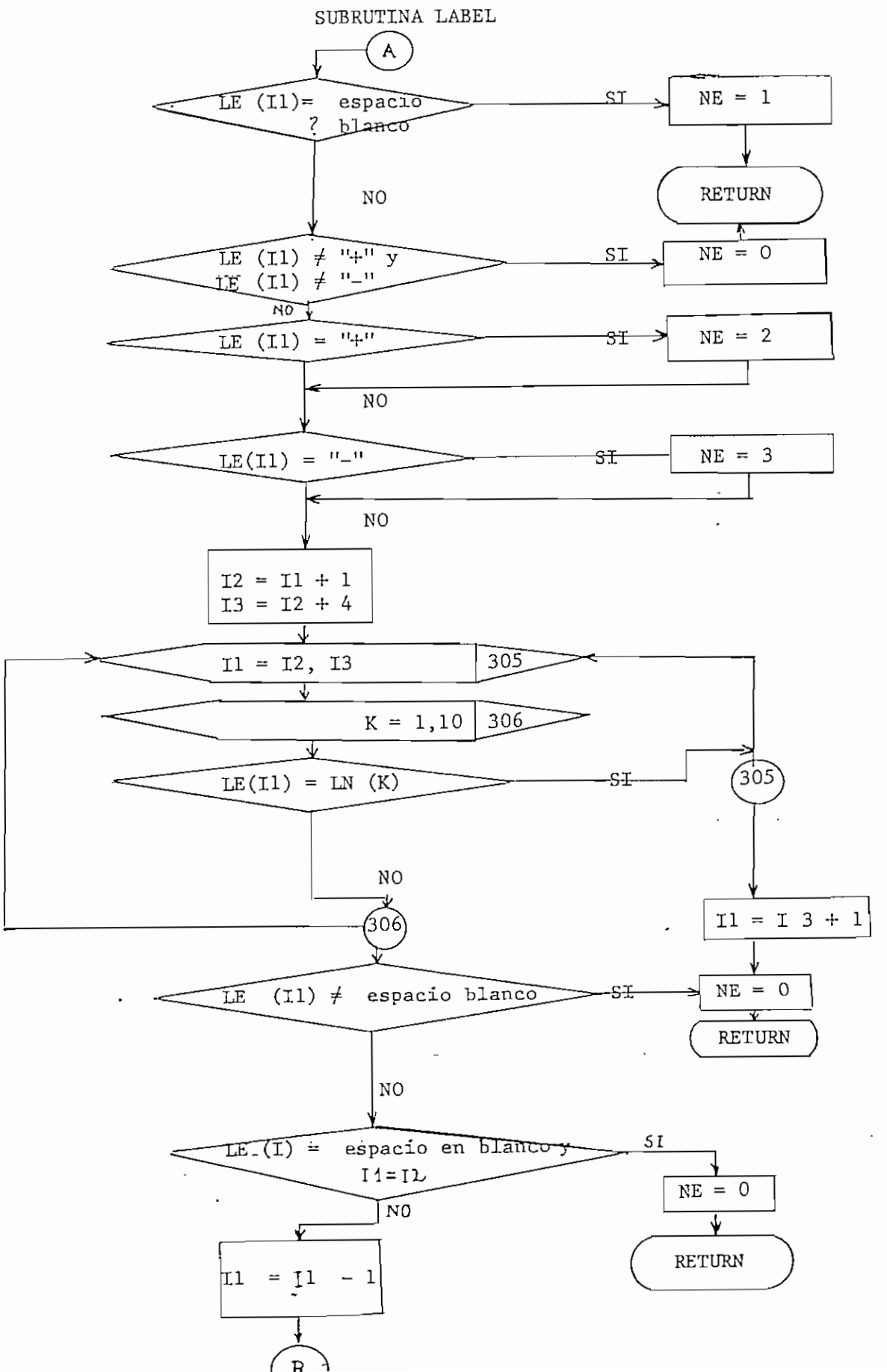
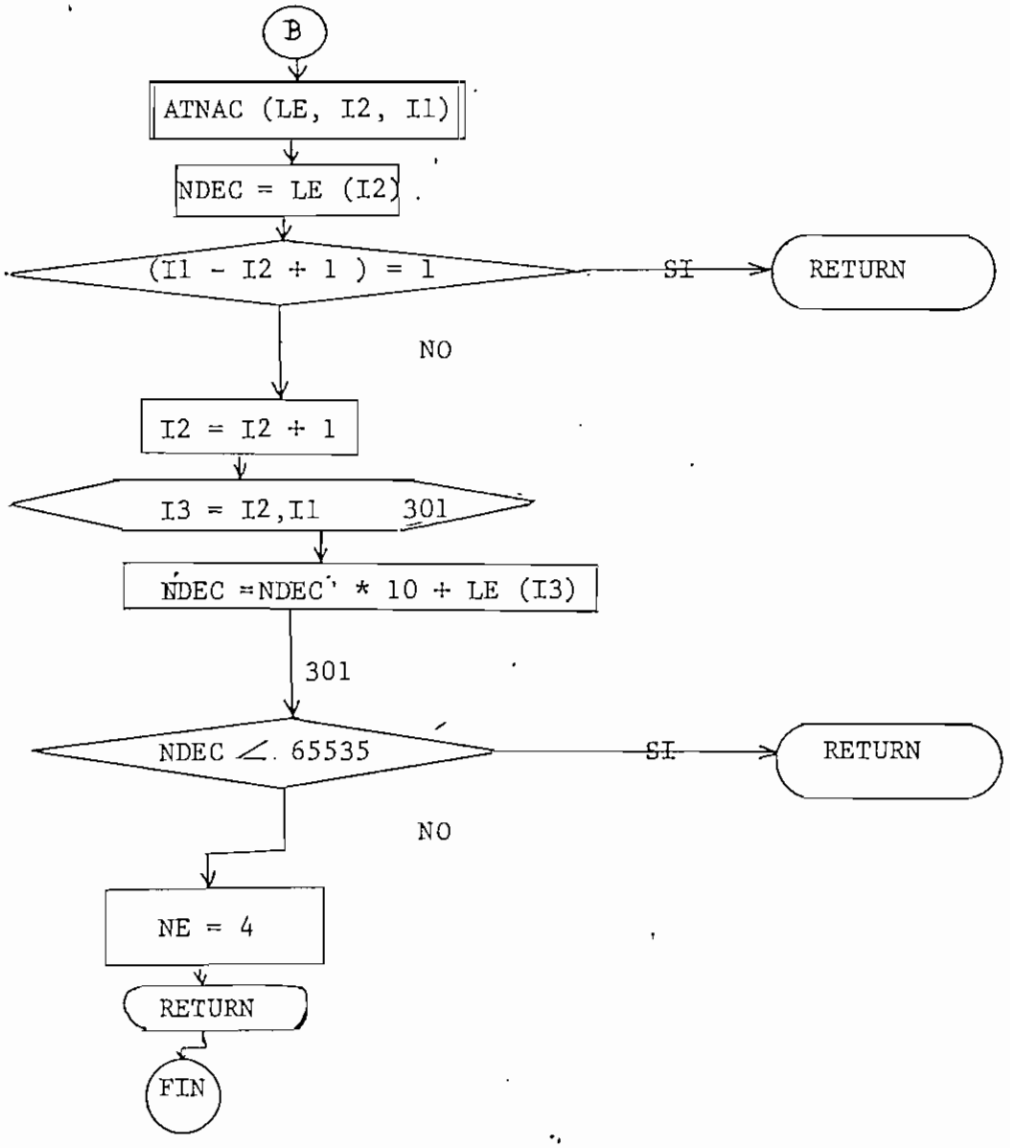

SUBRUTINA LABEL
A
LE (I1)= espacio ? blanco
SI
NE = 1
RETURN
NO
LE (I1) ≠ "+" y LE (I1) ≠ "-"
SI
NE = 0
NO
LE (I1) = "+"
SI
NE = 2
NO
LE(I1) = "-"
SI
NE = 3
NO
I2 = I1 + 1
I3 = I2 + 4
I1 = I2, I3
305
K = 1,10
306
LE(I1) = LN (K)
SI
305
NO
306
I1 = I 3 + 1
LE (I1) ≠ espacio blanco
SI
NE = 0
RETURN
NO
LE (I) = espacio en blanco y I1=I2
SI
NE = 0
RETURN
NO
I1 = I1 - 1
B

Fig. 4.15 SUBRUTINA LABEL



SUBRUTINA NTYPE

Definición : SUBRUTINA NTYPE (LSV,NC,NDEQ,NCM,NPP)

Propósito : Analiza si se tiene un número válido, determina su tipo y equivalente decimal.

Subprogramas llamados : Ninguno

Forma de utilización : CALL NTYPE (LSV, NC, NDER, NCM, NPP)

Parámetros : LSV, NC, NDEQ, NCM, NPP

LSV: Es el arreglo que se recibe para ser analizado por el subprograma.

NC : 0 - no número

1 - # binario

2 - # octal
3 - # decimal
4 - # hexadecimal

NDEQ: equivalente decimal del número

NCM : 1 - número seguido por un blanco

2 - número seguido por una coma

Datos de entrada : LSV

Datos de salida : NC, NDEQ, NCM, NPP

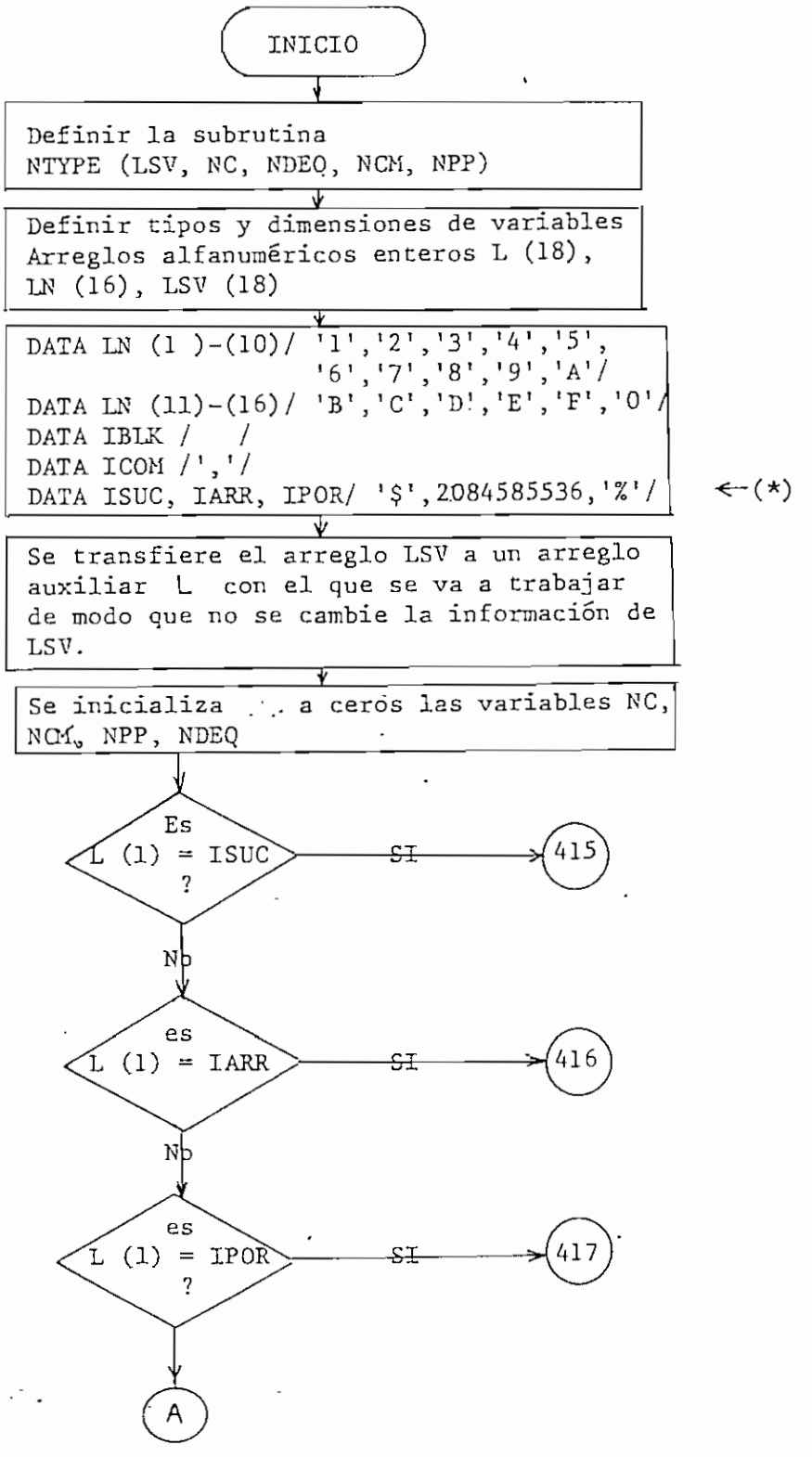
Diagrama de flujo : Ver Fig. 4. 16

Listado : En Apéndice B, pág. 258.

Funciones que realiza dentro del compilador:

Esta subrutina realiza análisis sintáctico, semántico y lexicográfico.

Fig. 4.16 SUBRUTINA NTYPE



(*)

NOTA: 2084585536, es la representación interna del signo de arrobas. Se la debe colocar, pues el sistema conversacional utilizado (CYTOS I), no acepta un símbolo de arrobas entre apóstrofes. Es decir que es equivalente colocar: 'a', ó 2084585536, pero la primera forma no es aœptada. Esta representación interna de ese carácter especial se obtiene con el programa ALFCD.

Fig. 4.16

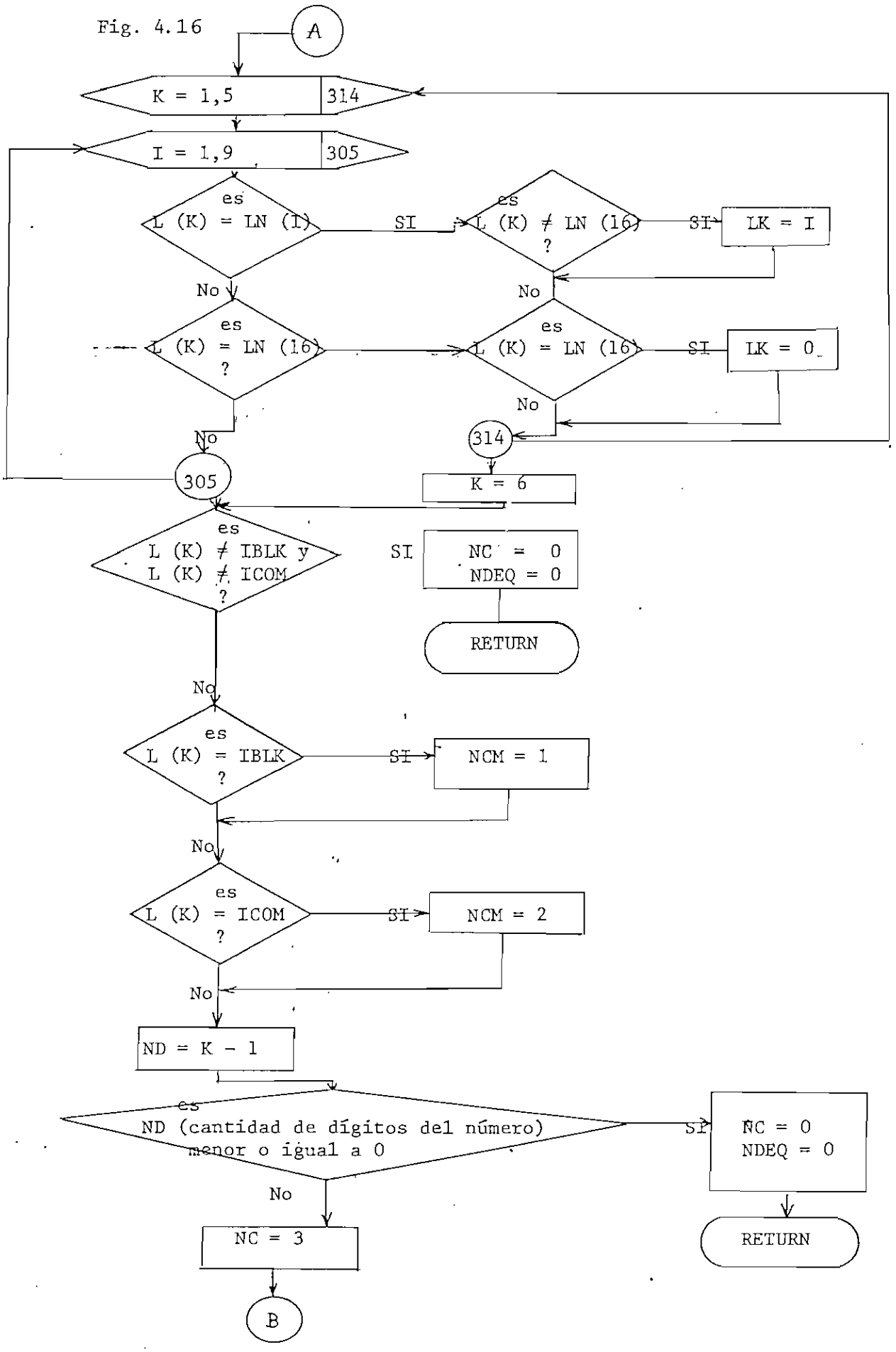

A
K = 1,5
314
I = 1,9
305
es L (K) = LN (I)
SI
es L (K) ≠ LN (16) ?
SI
LK = I
No
No
es L (K) = LN (16) ?
es L (K) = LN (16)
SI
LK = 0
No
No
314
305
K = 6
es L (K) ≠ IBLK y L (K) ≠ ICOM ?
SI
NC = 0
NDEQ = 0
RETURN
No
es L (K) = IBLK ?
SI
NCM = 1
No
es L (K) = ICOM ?
SI
NCM = 2
No
ND = K - 1
es ND (cantidad de dígitos del número) menor o igual a 0
SI
NC = 0
NDEQ = 0
RETURN
No
NC = 3
B

Fig. 4.16

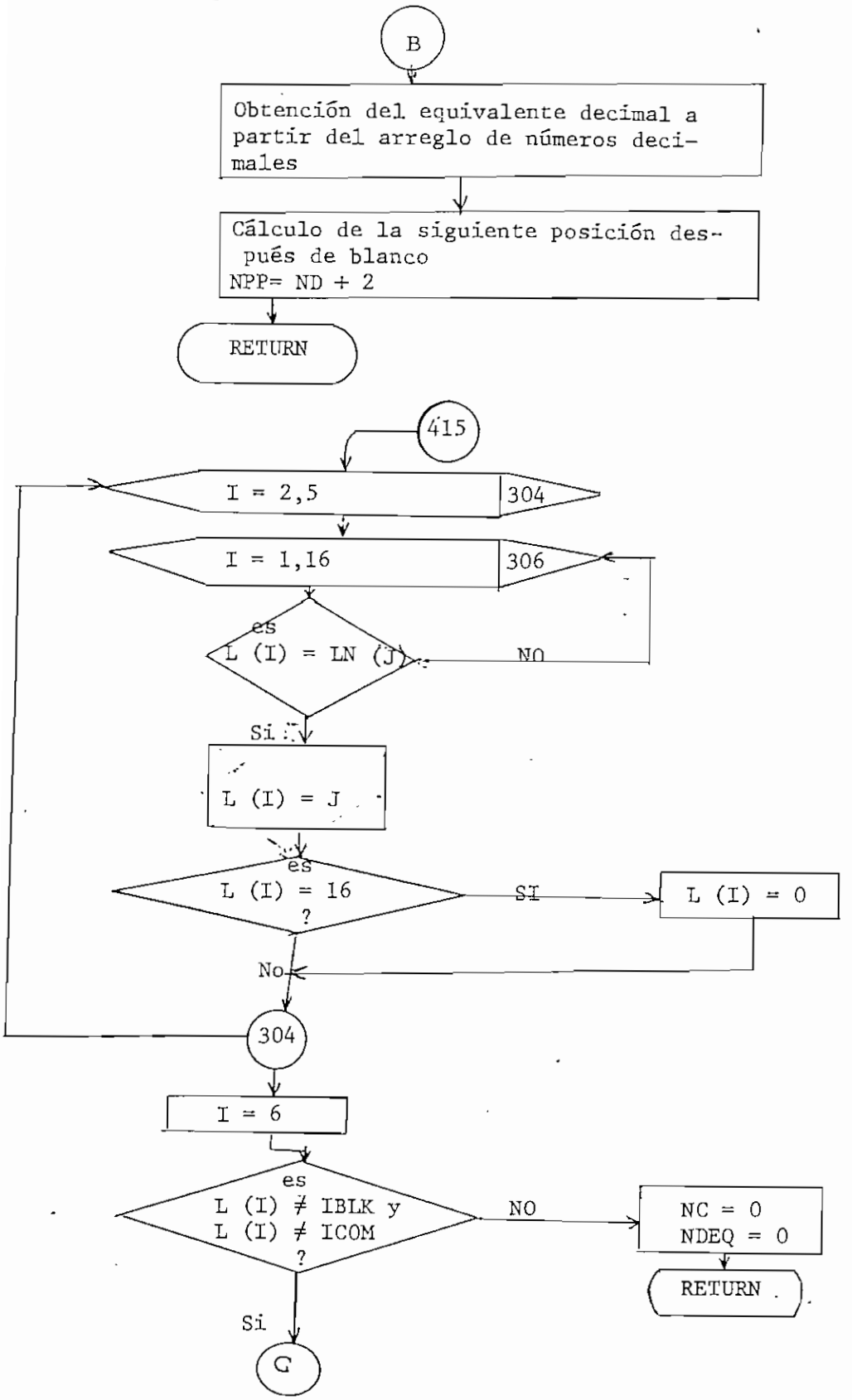
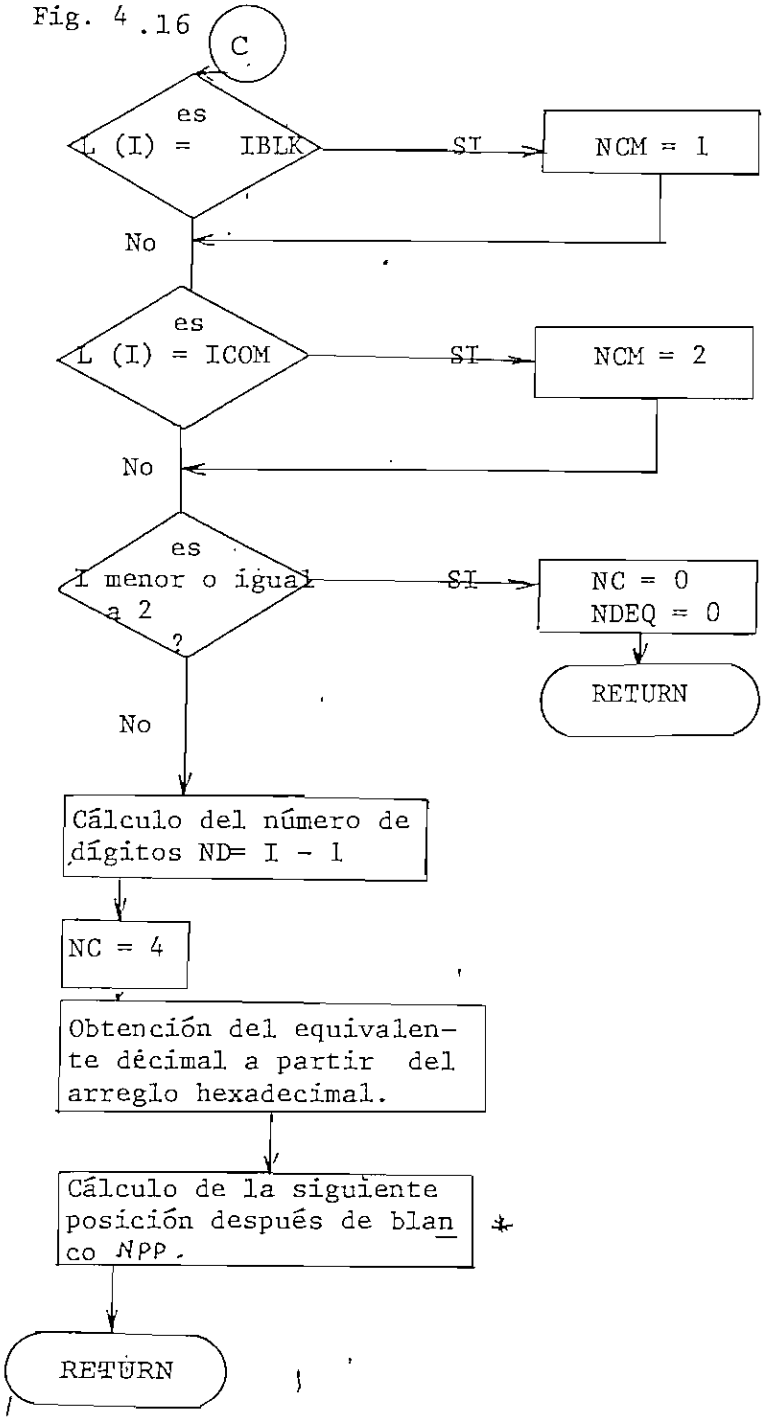

B
Obtención del equivalente decimal a partir del arreglo de números decimales
Cálculo de la siguiente posición después de blanco
NPP= ND + 2
RETURN
415
I = 2,5
304
I = 1,16
306
es
L (I) = LN (J)
NO
Si
L (I) = J
es
L (I) = 16
?
SI
L (I) = 0
No
304
I = 6
es
L (I) ≠ IBLK y
L (I) ≠ ICOM
?
NO
NC = 0
NDEQ = 0
RETURN
Si
G

Fig. 4.16



* NPP = ND + 2

Fig. 4.16

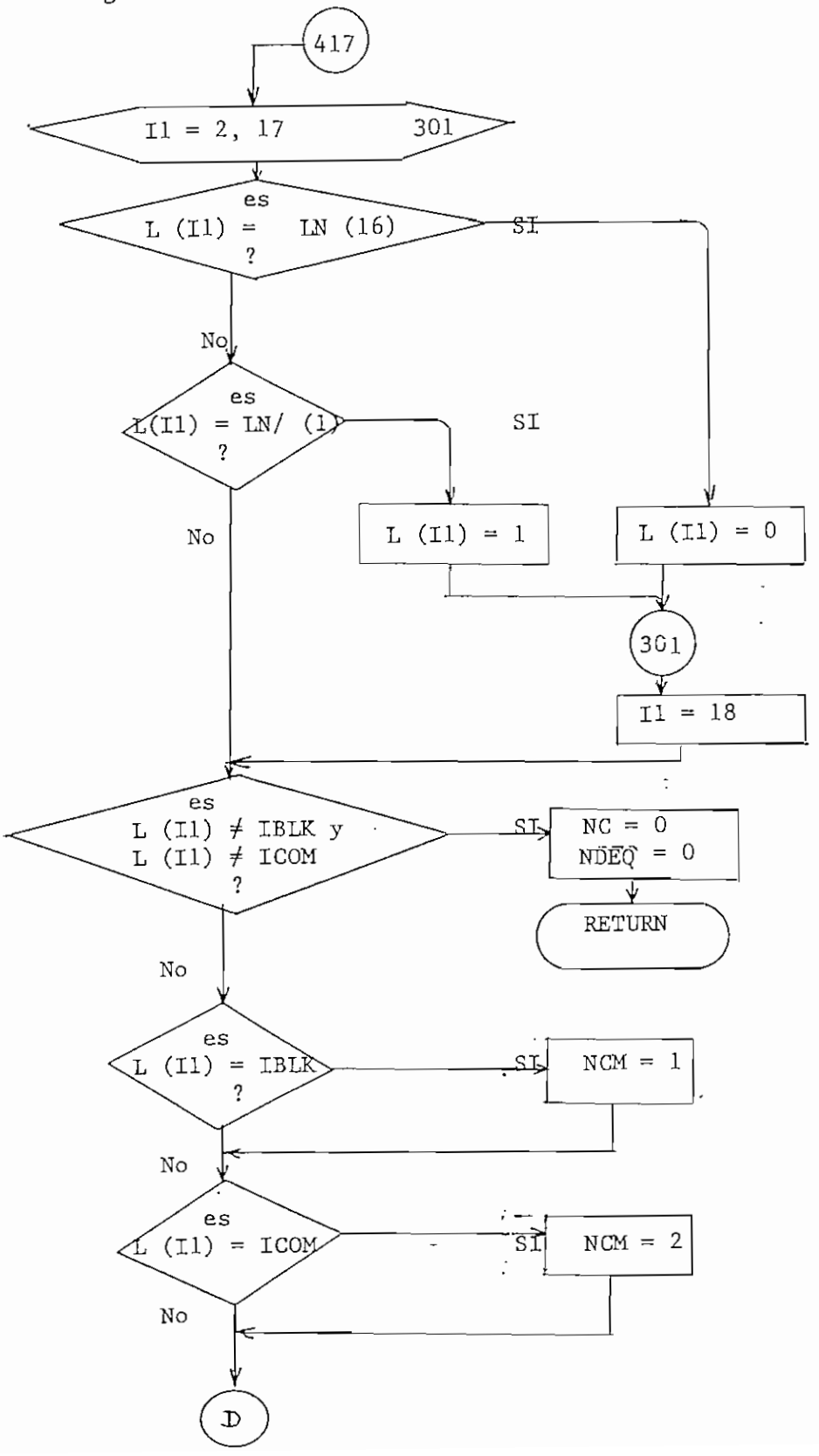
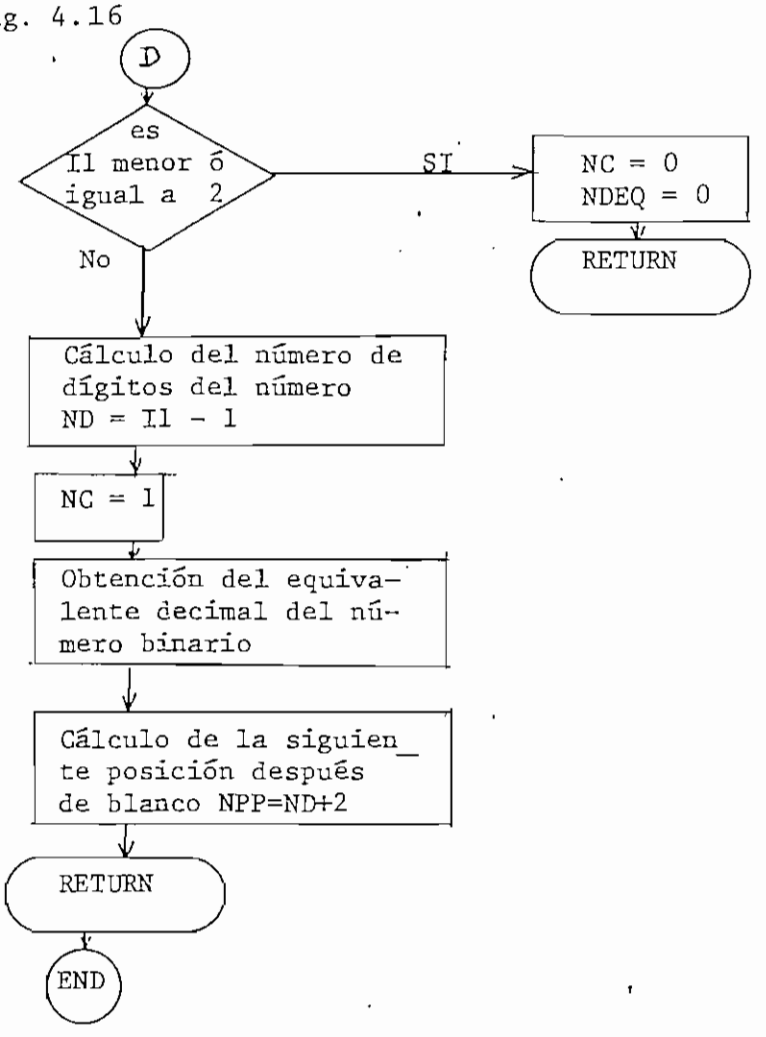

417
I1 = 2, 17
301
es
L (I1) = LN (16)
?
SI
No
es
L(I1) = LN/ (1)
?
SI
No
L (I1) = 1
L (I1) = 0
301
I1 = 18
es
L (I1) ≠ IBLK y
L (I1) ≠ ICOM
?
SI
NC = 0
NDEQ = 0
RETURN
No
es
L (I1) = IBLK
?
SI
NCM = 1
No
es
L (I1) = ICOM
SI
NCM = 2
No
D

Fig. 4.16


D
es
Il menor ó
igual a 2
SI
NC = 0
NDEQ = 0
RETURN
No
Cálculo del número de
dígitos del número
ND = Il - 1
NC = 1
Obtención del equiva-
lente decimal del nú-
mero binario
Cálculo de la siguien_
te posición después
de blanco NPP=ND+2
RETURN
END

SUBRUTINA ATNAC

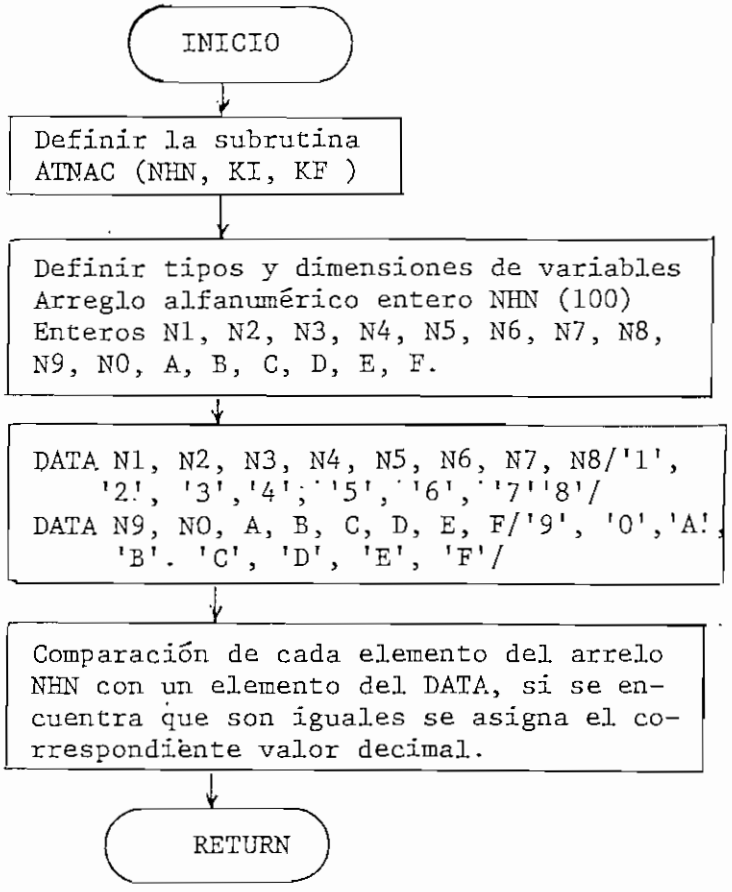
Definición	: SUBROUTINE ATNAC (NHN, KI, KF)
Propósito	: ATNAC-Conversión de alfanumérico a numérico
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL ATNAC (NHN, KI, KF)
Parámetros	: NHN, KI, KF
	NHN: arreglo de entrada conteniendo la información a ser convertida. Arreglo de salida conteniendo la información convertida.
	KI : Punto inicial del arreglo
	KF : Punto final del arreglo
Datos de entrada	: NHN, KI, KF
Datos de salida	: NHN
Diagrama de flujo	: Ver Fig. 4.17
Listado	: Ver Apéndice B, pág.277.

Funciones que realiza dentro del compilador:

Es un auxiliar que permite encontrar los equivalentes numéricos de caracteres en representación alfanumérica, sirve en la interrelación de las fases, obteniendo resultados parciales que deben ser transmitidos de una parte del compilador hacia otra.

Fig. 4.17

SUBRUTINA ATNAC



SUBRUTINA MOVER

Definición : SUBROUTINE MOVER (LA,NI,NF,LB,NII)

Propósito : Traslada elementos de un arreglo de una posición a otra.

Subprogramas llamados : Ninguno

Forma de utilización : CALL MOVER (LA,NI,NF,LB,NII)

Parámetros : LA , NI, NF, LB, NII

LA : arreglo inicial entregado a la subrutina

LB : arreglo final con elementos trasladados

NI : Posición inicial

NF : Posición final

NII: Posición inicial de los elementos movidos en el otro arreglo.

Datos de entrada : LA, NI, NF, NII

Datos de salida : LB

Diagrama de flujo : Fig. 4 .18

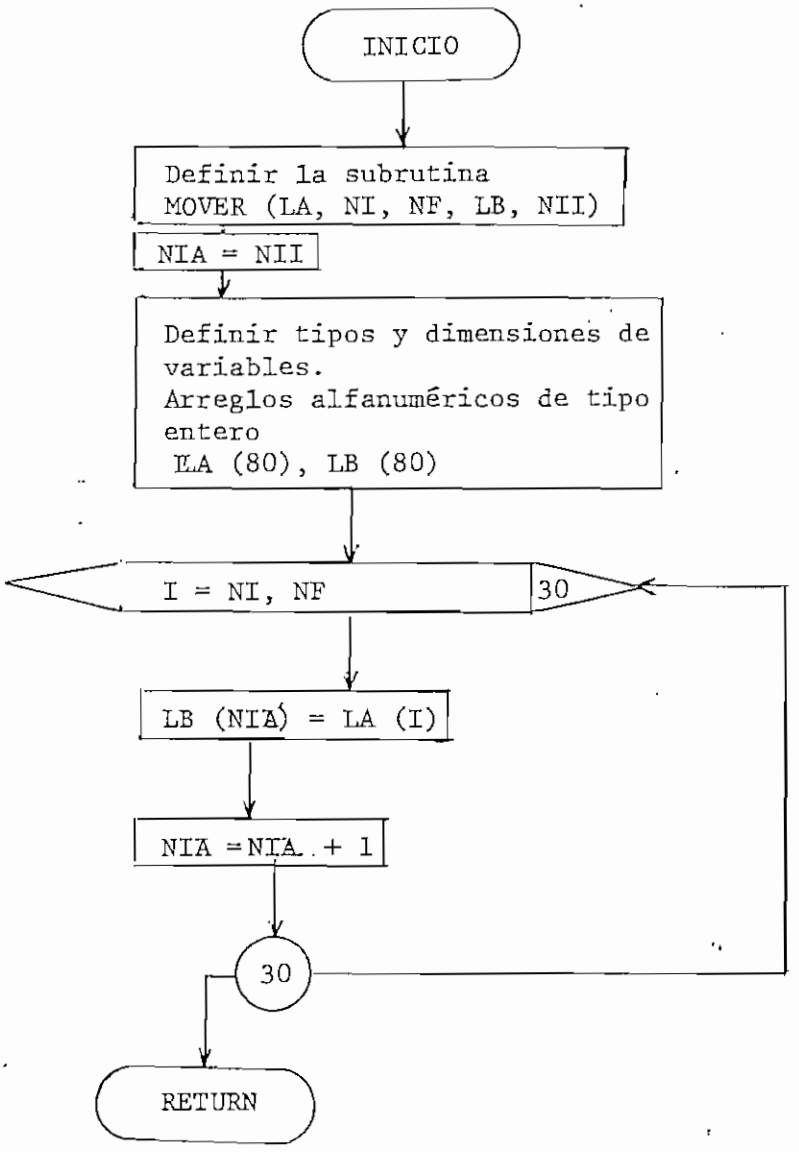
Listado : Ver apéndice B, pág.256 .

Funciones que realiza dentro del compilador:

Sirve en el análisis lexicográfico, pues separa una instrucción completa de entrada en algunos componentes de interés ubicados en nuevos arreglos.

Fig. 4.18

SUBRUTINA MOVER



SUBRUTINA IMPLI

Definición : SUBROUTINE IMPLI (LSV, NC, LR, NX)

Propósito : Reconocimiento de direccionamiento implícito.

Subprogramas llamados : ninguno

Forma de utilización : CALL IMPLI (LSV, NC, LR, NX)

Parámetros : LSV, NC, LR, NX

LSV: Operando de hasta 18 posiciones

NC : parámetro de la subrutina NTYPE

NX : - 5 direccionamiento implícito

- 6 no es direccionamiento implícito

LR : registro de 16 elementos del archivo NMONI que contiene el operando y sus respectivos códigos hexadecimales.

Datos de entrada : LSV, NC, LR.

Datos de salida : NX

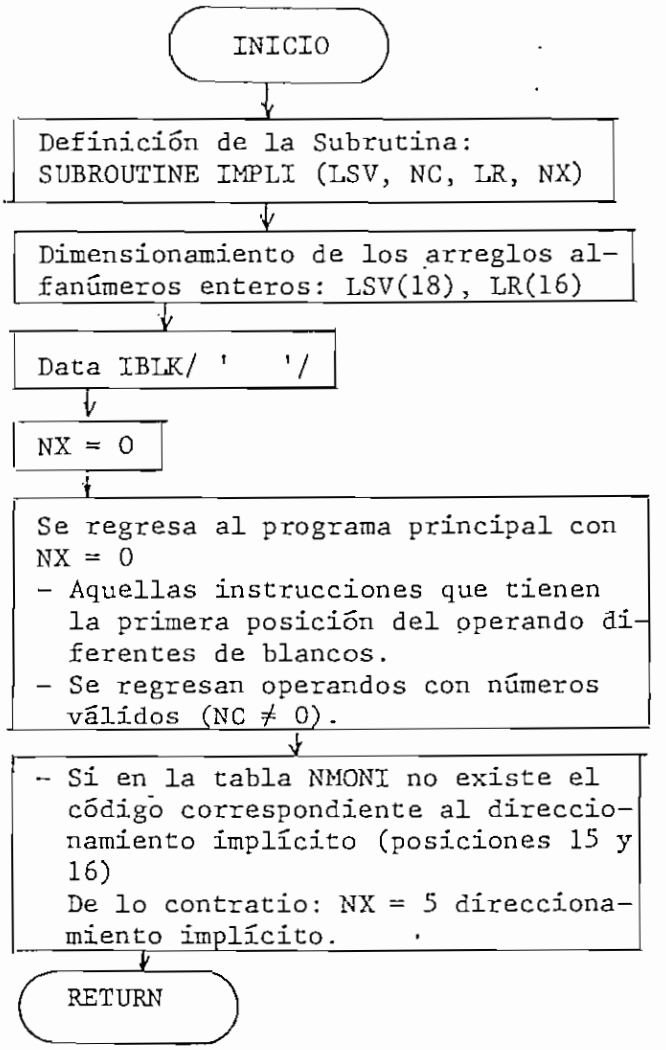
Diagrama de flujo : Fig. 4.19

Listado : Ver apéndice B, pág. 267 .

Funciones que realiza dentro del compilador:

Realiza una parte del análisis semántico, pues según la posición y combinación de identificadores válidos, determina el tipo de direccionamiento.

Fig. 4.19 SUBRUTINA IMPLI



SUBRUTINA VNMOP

Definición : SUBROUTINE VNMOP (L80, LCOD, NER)

Propósito : Verifica si existe en esa línea un código NMOTÉCNICO válido.

Subprogramas llamados : ninguno

Forma de utlización : CALL VNMOP (L80, LCOD, NERO)

Parámetros : L80, LCOD, NER

L80: línea de programa de 80 posiciones

LCOD: línea del archivo 1 NMONI que contiene un código nmotécnico y los hexadecimales correspondientes

NER: - 0 no existe código nmotécnico válido
- 1 si existe código nmotécnico válido

Datos de entrada : L80

Datos de salida : LCOD, NER

Diagrama de flujo : Fig. 4.20

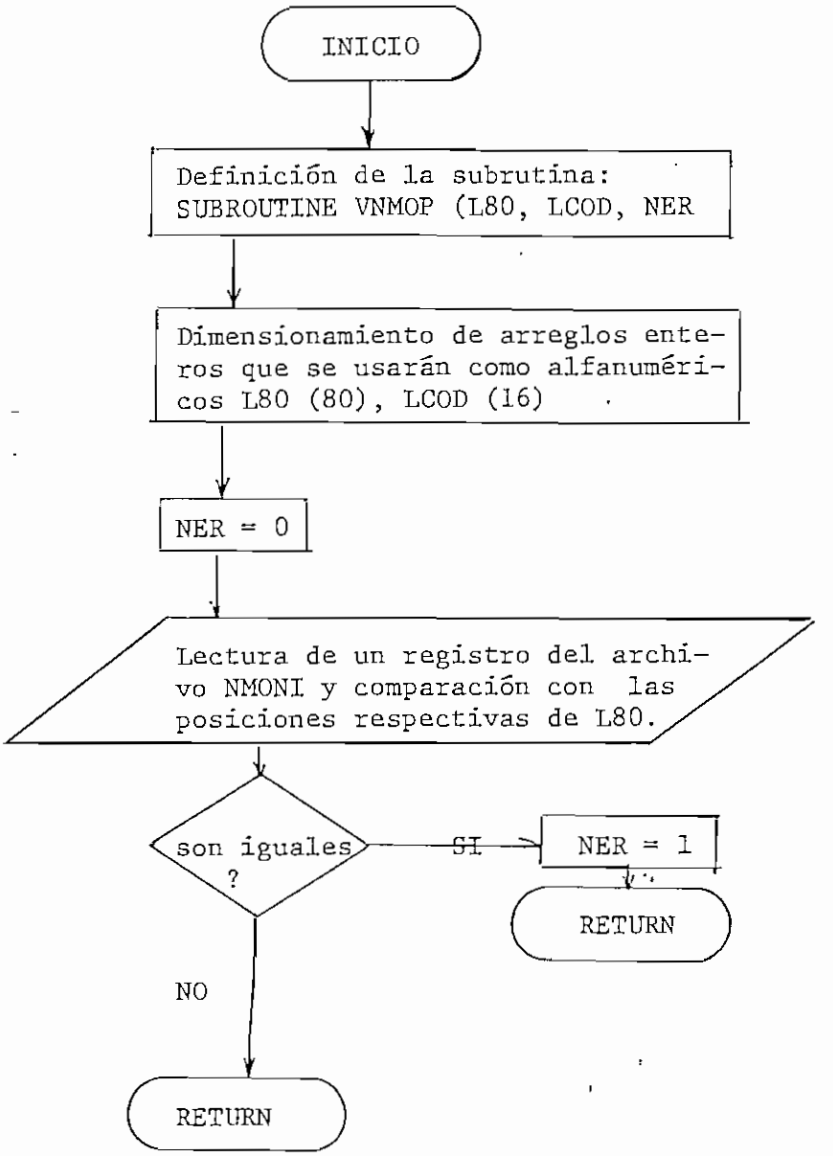
Listado : Ver Apéndice B, pág.257

Funciones que desempeña dentro del compilador:

Es un auxiliar en la relación entre el assembler y la tabla de símbolos (archivo NMONI)

Fig. 4.20

SUBRUTINA VNMOP



SUBRUTINA HRNTF

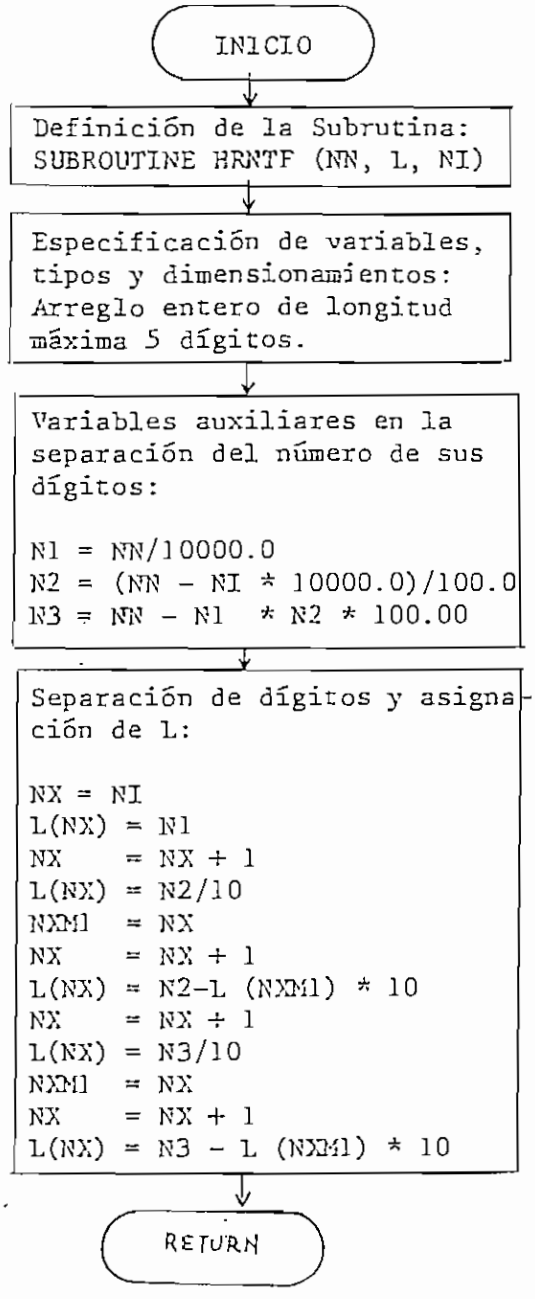
Definición	:	SUBROUTINE HRNTF (NN, L, NI)
Propósito	:	Subrutina que separa un número entero NN en los dígitos que lo componen y los coloca dentro del arreglo L a partir del elemento NI.
Subprogramas llamados	:	Ninguno
Forma de utilización	:	CALL HRNTF (NN, L, NI)
Parámetros	:	NN, L, NI
		NN : Número entero entregado a la Subrutina.
		L : Arreglo en el que se va a colocar los dígitos.
		NI : Posición desde la cual se van a ubicar los dígitos del número entero NN.
Datos de entrada	:	NN, L, NI.
Datos de salida	:	L, arreglo modificado
Diagrama de flujo	:	Fig. 4.21
Listado	:	Ver Apéndice B, pág. 279 .

Funciones que realiza dentro del compilador:

Este Subprograma se lo utiliza como un auxiliar para emitir la generación de código.

Fig. 4.21

SUBRUTINA HRNTF



SUBRUTINA DIREC

Definición	: SUBROUTINE DIREC (NDEN, NE, NC, NX)
Propósito	: Reconocimiento de direccionamientos directo y extendido.
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL DIREC (NDEN, NE, NC, NX)
Parámetros	: NDEN, NE, NC, NX
	NDEN, NC: parámetro de la subrutina NTYPE (pág.).
	NE : Parámetro de la subrutina LABEL (pág.).
	NX : - 2 direccionamiento directo
	NX : - 4 direccionamiento extendido
	NX : - 0 ningún tipo de direccionamiento.
Datos de entrada	: NDEN, NE, NC
Datos de salida	: NX
Diagrama de flujo	: Fig. 4.22
Listado	: Ver Apéndice B, pág.266 .

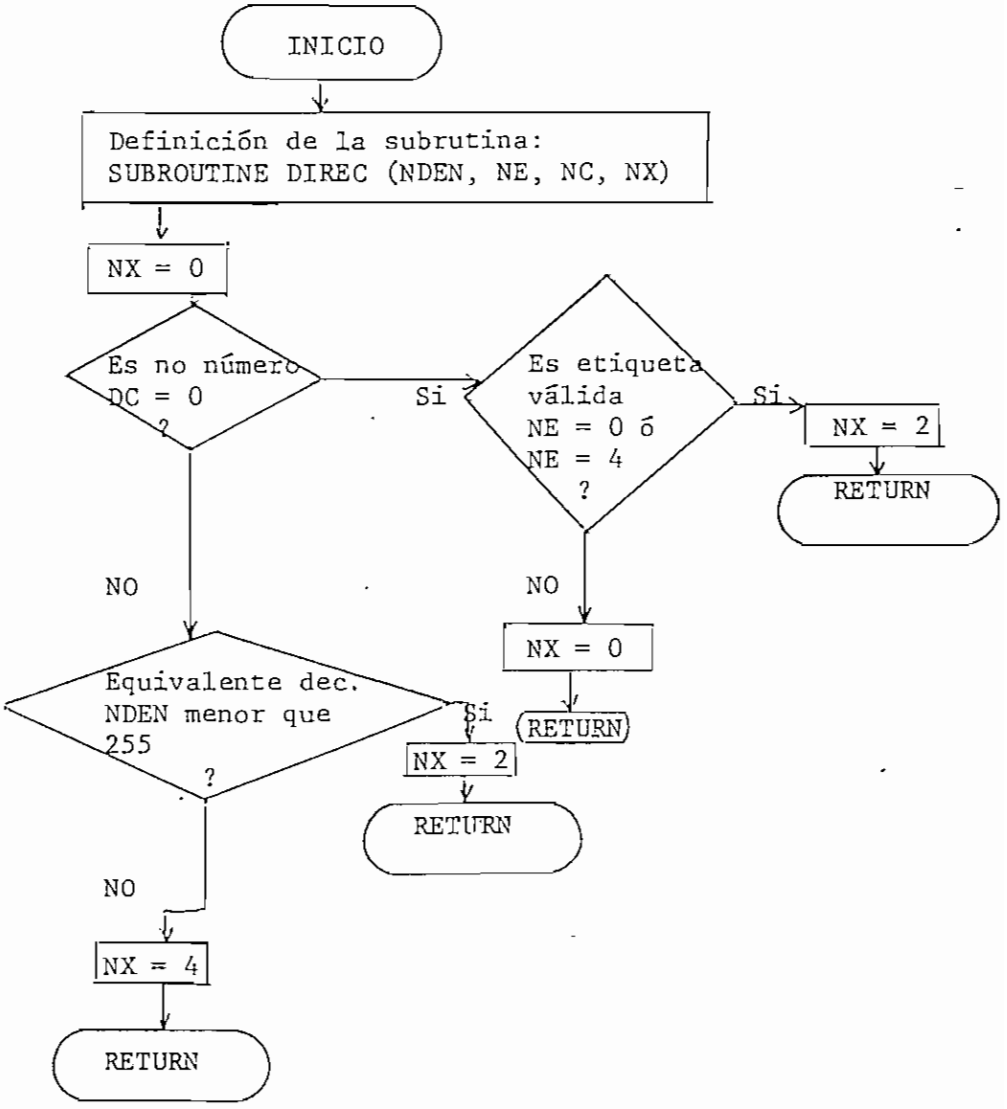
Funciones que realiza dentro del compilador:

Realiza una parte del análisis semántico, pues determina el tipo de direccionamiento según la combinación de identificadores válidos.

En esta subrutina se reconoce direccionamientos directo y extendido.

Se diferencia entre estos dos tipos en caso de direccionamientos con operandos numéricos, en cambio cuando se tiene etiquetas no se puede saber cuál de los dos direccionamientos es, sino después cuando se relaciona la etiqueta con su valor. Esto se hace en la subrutina COD24, donde se determinará para este caso si el direccionamiento es directo o extendido.

Fig. 4.22 SUBRUTINA DIREC



SUBRUTINA INDEX

Definición	: SUBROUTINE INDEX (LSV, NC, NDEQ., NCM, NPP, NX)
Propósito	: Subrutina que reconoce direccionamiento indexádo.
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL INDEX (LSV, NC, NDEQ, NCM, NPP,NX)
Parámetros	: LSV, NC, NDEQ, NCM, NPP, NX LSV- Operando NC, NDEQ, NCM, NPP - párámetros de la subrutina NTYPE (Pag.). NX : -3 direccionamiento indexado NX : -0 no direccionamiento indexádo.
Datos de entrada	: LSV, NC, NDEQ, NCM, NPP
Datos de salida	: NX
Diagrama de flujo	: Fig. 4.23
Listado	: ver apéndice B, pág.265 .

Funciones que realiza dentro del compilador:

Es parte del análisis semántico de reconocimiento de tipo de instrucciones a partir del análisis de la combinación de identificadores válidos.

El reconocimiento de este tipo de direccionamiento se lo va a realizar para ejemplificar las tablas de tran-

sición que se estudiaron en el 2.4.2 del presente trabajo. De esta manera en esta fase se logra una mayor velocidad, pero a costa de un incremento en ocupación de memoria.

Tabla $4._2$

Tabla de transición para direccionamiento indexado.

Estados : 1 - estado inicial

2 - número seguido por una coma

3 - X

4 - blanco

	No.	X	b	⊣
1	2	3	0	0
2	0	3	0	0
3	0	0	4	0
4	0	0	0	1

Fig. 4 .23

SUBRUTINA INDEX

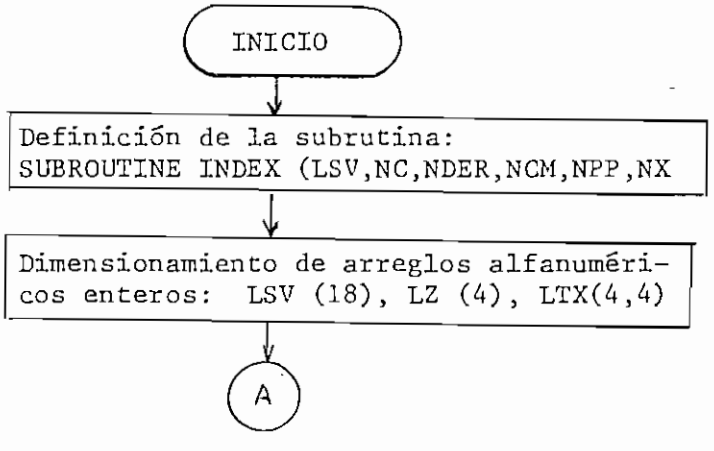


Fig. 4.23

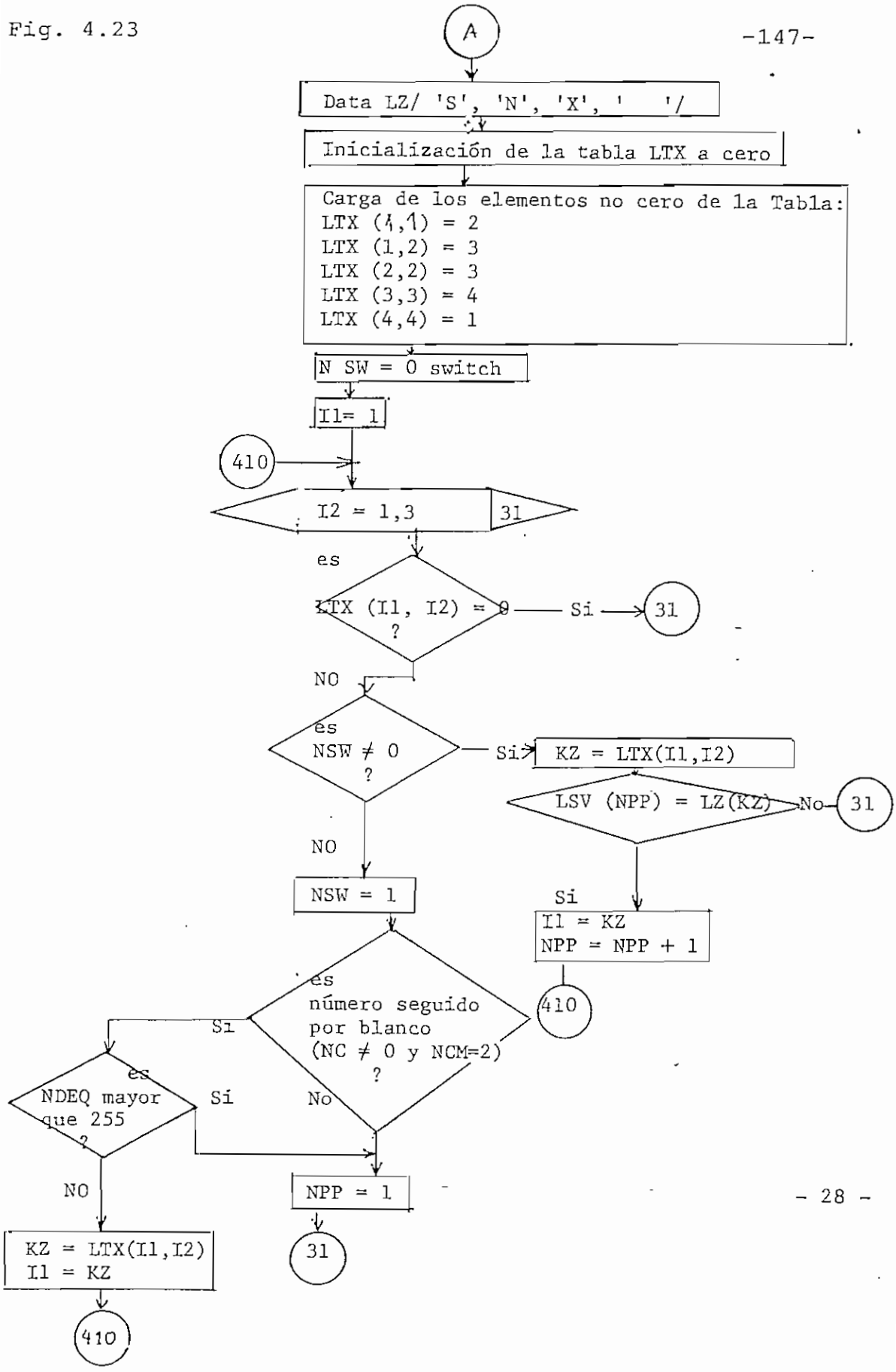
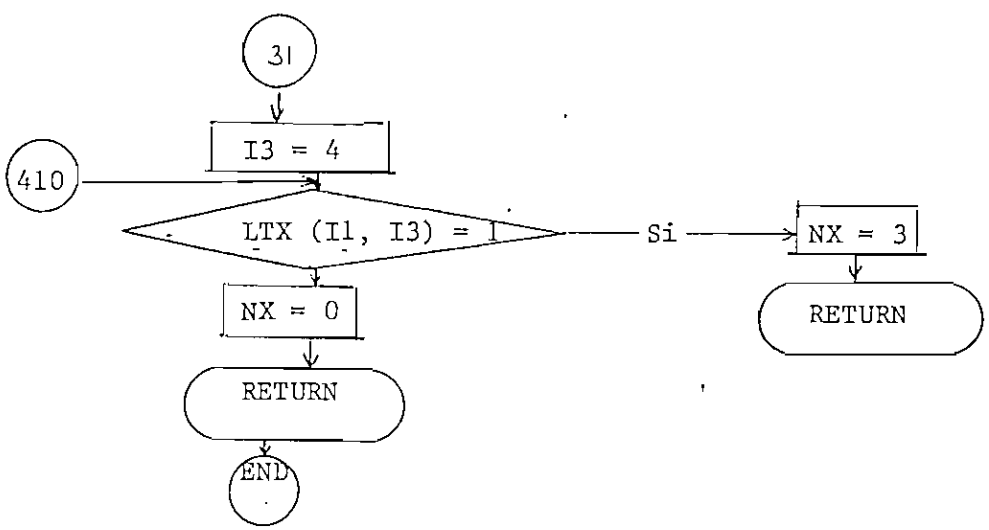

A
Data LZ/ 'S', 'N', 'X', ' '/
Inicialización de la tabla LTX a cero
Carga de los elementos no cero de la Tabla:
LTX (1,1) = 2
LTX (1,2) = 3
LTX (2,2) = 3
LTX (3,3) = 4
LTX (4,4) = 1
N SW = 0 switch
I1= 1
410
I2 = 1,3
31
es
LTX (I1, I2) = 0 ?
Si
31
NO
es
NSW ≠ 0 ?
Si
KZ = LTX(I1,I2)
LSV (NPP) = LZ(KZ)
No
31
NO
NSW = 1
Si
I1 = KZ
NPP = NPP + 1
410
es
número seguido por blanco
(NC ≠ 0 y NCM=2)
?
Si
No
es
NDEQ mayor que 255
?
Si
NO
NPP = 1
31
KZ = LTX(I1,I2)
I1 = KZ
410

Fig. 4.23



Como se puede observar en el desarrollo de la Subrutina INDEX para reconocimiento de direccionamiento Indexado, el problema se vuelve más complicado con el uso de las tablas de transición. La programación tiene mayores detalles que debe tomar en cuenta.

Para un programador existen maneras más obvias de enfrentar el análisis del asunto. Además el consumo de localidades de memoria es mas grande pues el programa es más largo y se debe llenar una tabla que ocupa un arreglo de dimensiones 4x4.

Existen ventajas que se contraponen a lo anteriormente expresado como son las siguientes:

- Facilidad de hacer modificaciones de la gramática de reconocimiento con sólo cambiar el contenido de las tablas y hacer pequeños cambios al programa.

- La velocidad es mayor. El computador procesa más rapidamente el programa realizado en base a las tablas.

Se estudiará a continuación otro ejemplo de la utilización de la Teoría de Autómatas en el diseño de programas para reconocimientos de gramática. En este caso la subrutina a servirá para identificación de direccionamiento inmediato.

SUBRUTINA INMED

Definición : SUBROUTINE INMED (LSV, NX)

Propósito : Se encarga de considerar si la línea contiene una instrucción con direccionamiento inmediato.

Subprogramas llamados : NTYPE

Forma de utilización : CALL INMED (LSV, NX)

Parámetros : LSV, NX

LSV: operando guardado en un arreglo de 18 posiciones.

NX : - 0 no es direccionamiento inmediato
- 1 direccionamiento inmediato

Datos de entrada : LSV

Datos de salida : NX

Diagramas de flujo : Fig, 4.24

Listado : Ver Apéndice B, pág. 263.

Funciones que realiza dentro del compilador:

En la programación de esta subrutina y para dar otro ejemplo de las tablas de transición estudiadas en el Capítulo II, se va a configurar la subrutina en base a la siguiente tabla:

TABLA 4.3 :
Tabla de transición de direccionamiento inmediato

Signo '#'	2
'N' número	3
'b' blanco	4

	2	3	4	⊣
estado inicial	2			0
2		3		0
3			4	0
4				1

Fig. 4.24

SUBRUTINA INMED

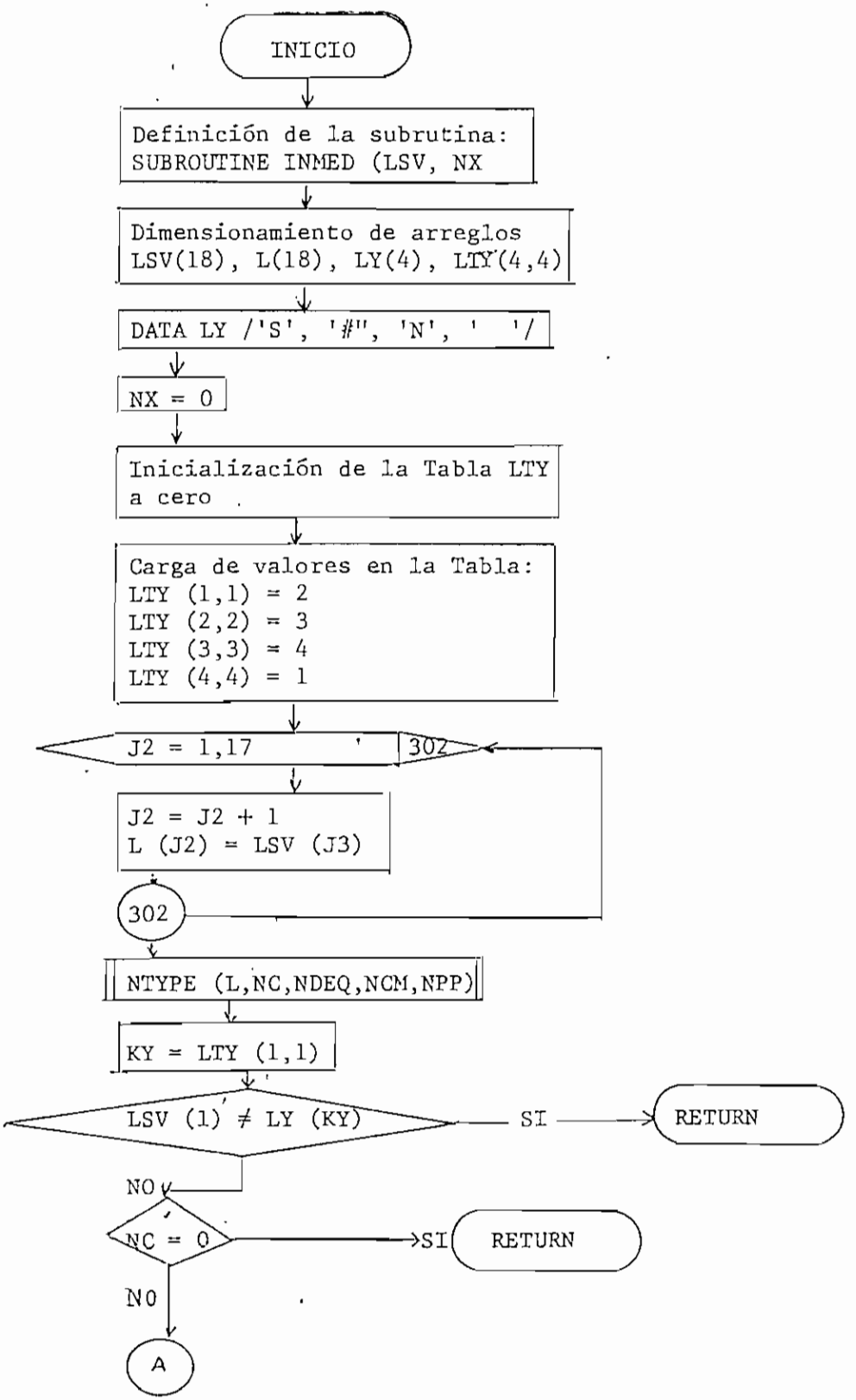
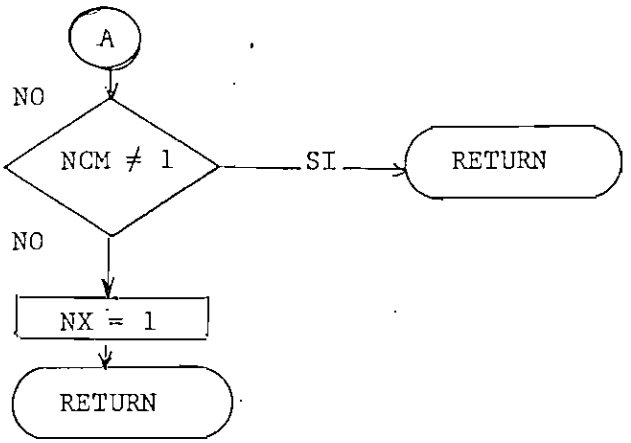


Fig. 4.24

SUBRUTINA INMED



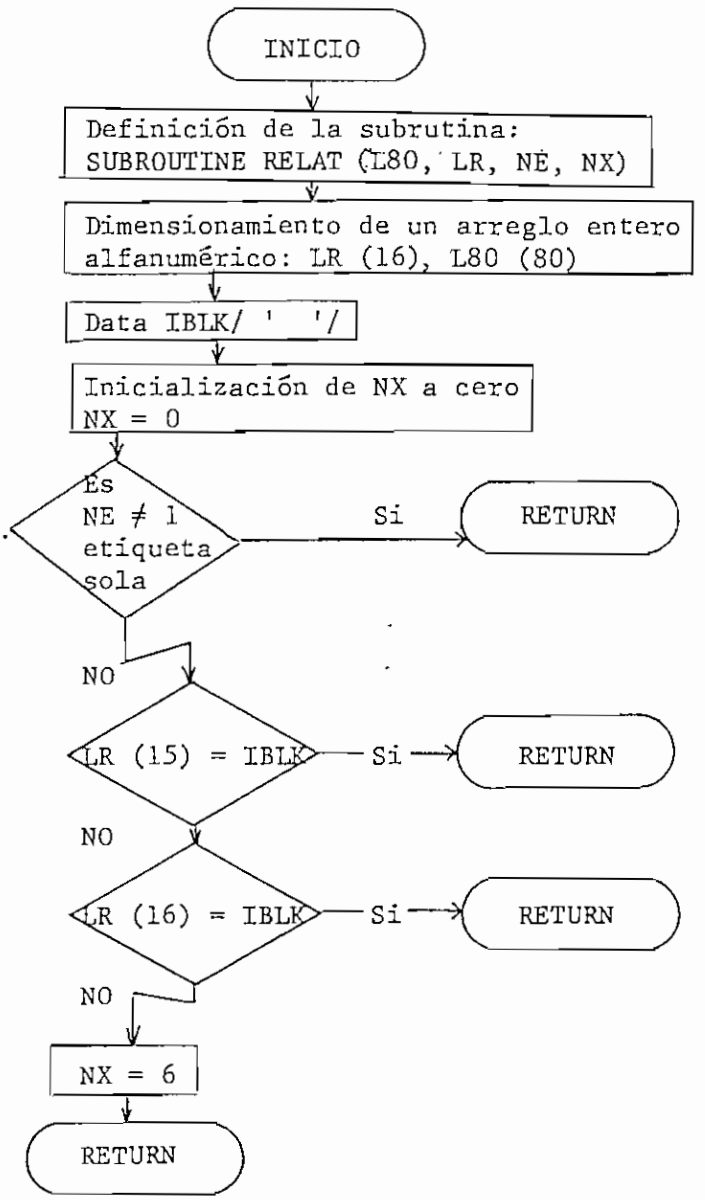
SUBRUTINA RELAT

Definición	: SUBROUTINE RELAT (L80, LR, NE, NX)
Propósito	: Subrutina de reconocimiento de direccionamiento relativo
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL RELAT (L80, LR, NE, NX)
Parámetros	: LR, NE, NX, L80
	LR : arreglo de 16 posiciones en las que se encuentran los códigos nmotécnico de las operaciones y sus respectivos hexadecimales.
	NE : parámetro de la subrutina LABEL (pág.).
	NX : 6 direccionamiento relativo
	0 no direccionamiento relativo
	L80: Arreglo de línea de programa
Datos de entrada	: LR, NE , L80
Datos de salida	: NX
Diagrama de flujo	: Fig. 4.25
Listado	: ver Apéndice B, pág. 264.

Funciones que realiza dentro del compilador:

Reconoce un tipo de direccionamiento de acuerdo a las combinaciones de identificadores válidos que éste puede aceptar, por lo tanto, se lo puede considerar como un lemento de la fase de análisis semántico.

Fig. 4..25 SUBRUTINA RELAT



SUBRUTINA VDF

Definición : SUBROUTINE VDF (L80, NTY)

Propósito : Subrutina que analiza si se tiene una pseudo-operación válida.

Subprogramas llamados : DIR

Forma de utilización : CALL VDF (L80, NTY)

Parámetros : L80, NTY

L80: arreglo en el que se encuentra la instrucción
NTY - 0 no hay pseudo-operación válida
- 1 hay pseudo-operación ORIGEN (ORG)
- 2 hay pseudo-operación EQUAL (EQU)
- 3 hay pseudo-operación RESERVE-MEMORY-BY TES (RMB)
- 4 hay pseudo-operación NAME (NAM)
- 5 hay pseudo-operacion END (END)
- 6 hay pseudo-operación FORM-CONSTANT-BYTE
- 7 hay pseudo-operación FORM-CONSTANT-CHARACTER

Datos de entrada. : L80.

Datos de salida : NTY

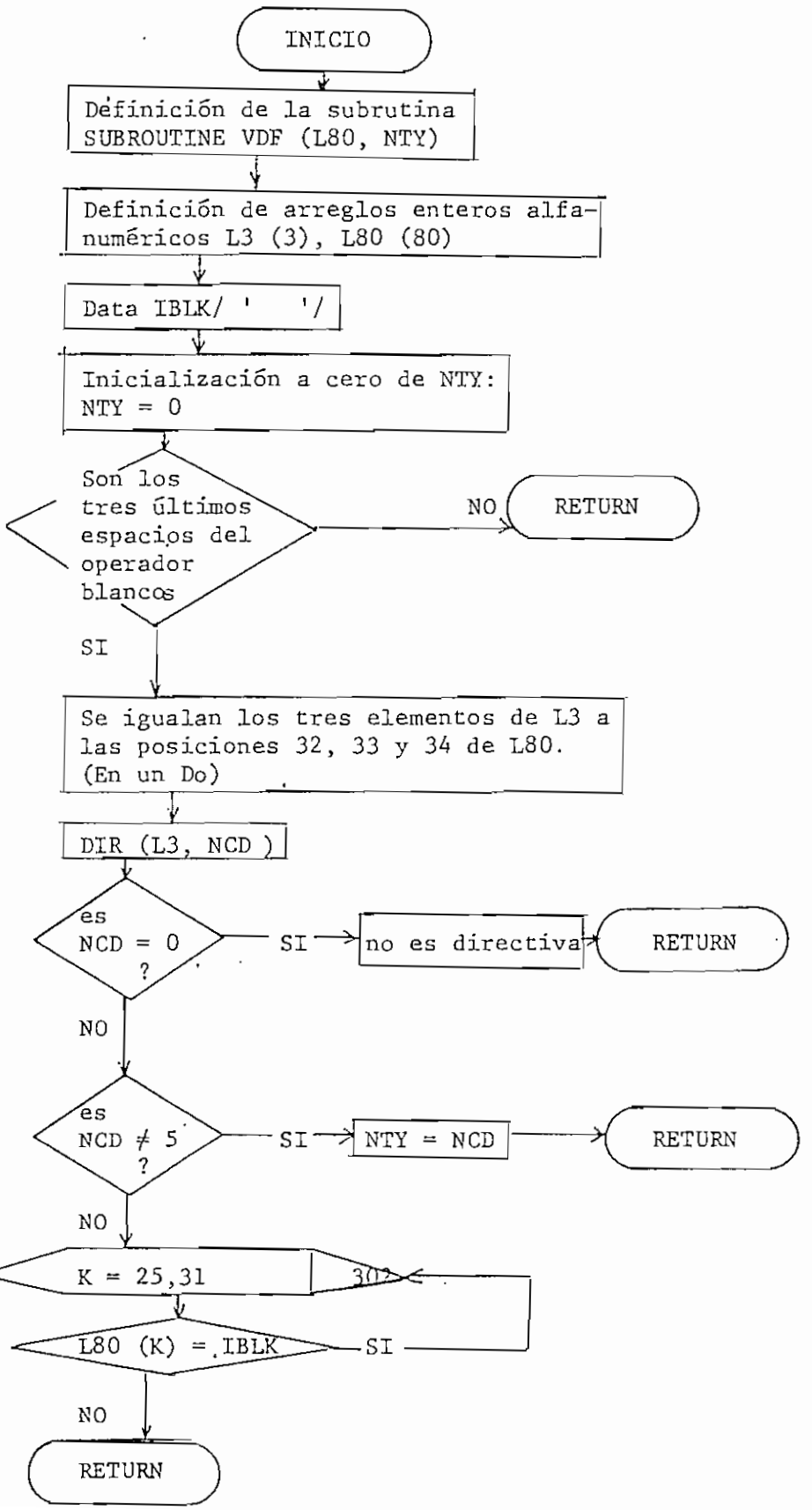
Diagramas de flujo : Fig. 4.26

Listado : Ver Apéndice B, pág. 248 .

Funciones que realiza dentro del compilador:

Analiza la validez de pseudo-operaciones, por lo tanto realiza análisis sintáctico (validez de identificadores).

Fig. 4.26 SUBRUTINA VDF



SUBRUTINA DIR

Definición : SUBROUTINE DIR (L, NCD)

Propósito : Determina de qué pseudo-operación se trata

Subprogramas llamados : CMP

Forma de utilización : CALL DIR (L, NCD)

Parámetros : L, NCD

L : Nombre de la pseudo-operación

NCD: - 0 no pseudo-operación

- 1 pseudo-operación ORG

- 2 pseudo-operación EQU

- 3 pseudo-operación RMB

- 4 pseudo-operación NAM

- 5 pseudo-operación END

- 6 pseudo-operación FCC

- 7 pseudo-operación FCB

Datos de entrada : L

Datos de salida : NCD

Diagrama de flujo : Fig. 4.27

Listado : ver Apéndice B, pág. 280.

Funciones que realiza dentro del compilador:

Especifica el tipo de una pseudo-operación válida, por lo tanto hace una parte del análisis semántico.

Fig. 4.27 SUBRUTINA DIR

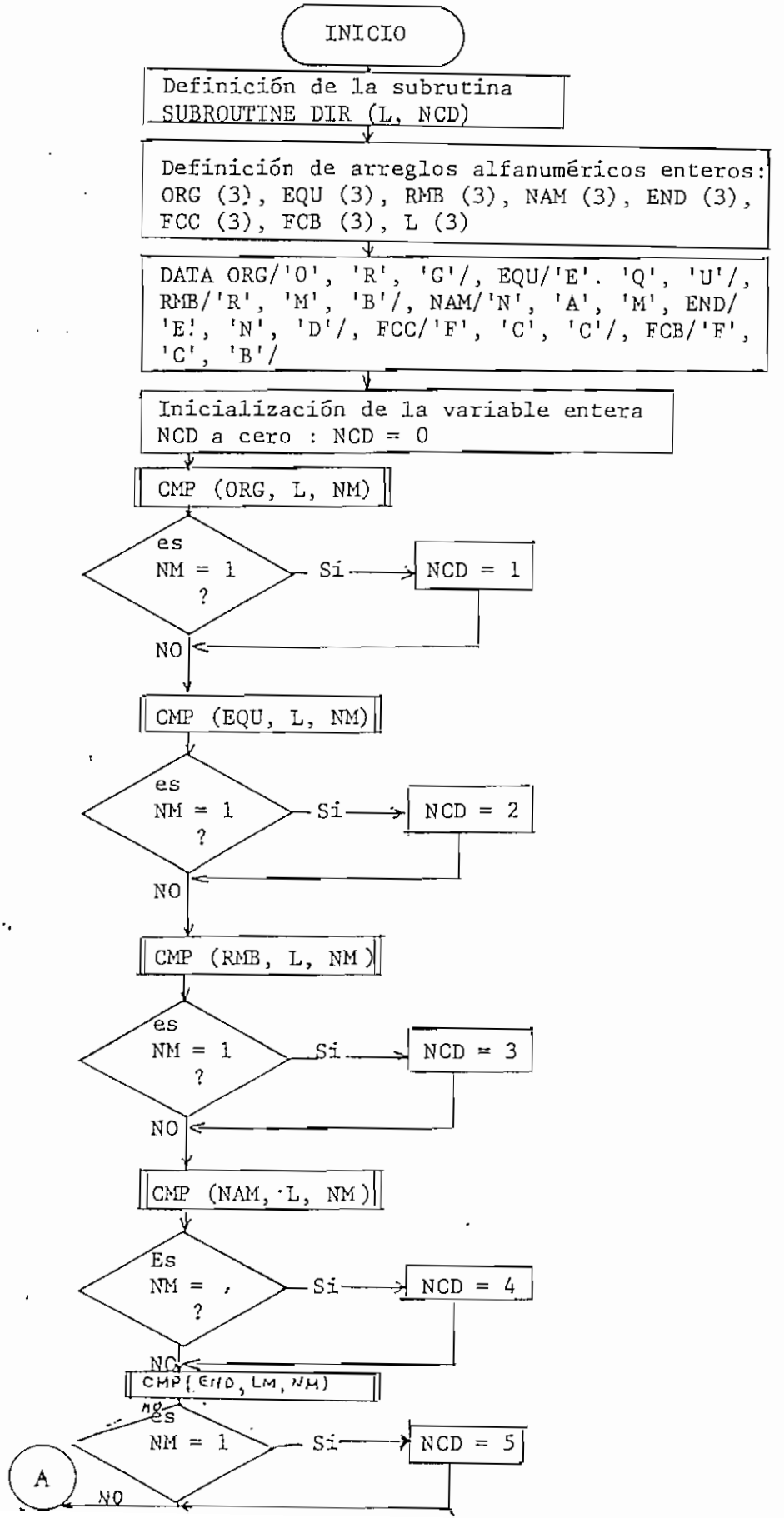
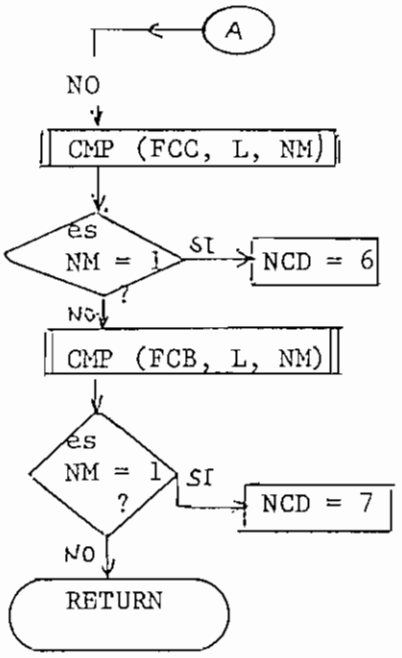


Fig. 4.27 SUBRUTINA DIR



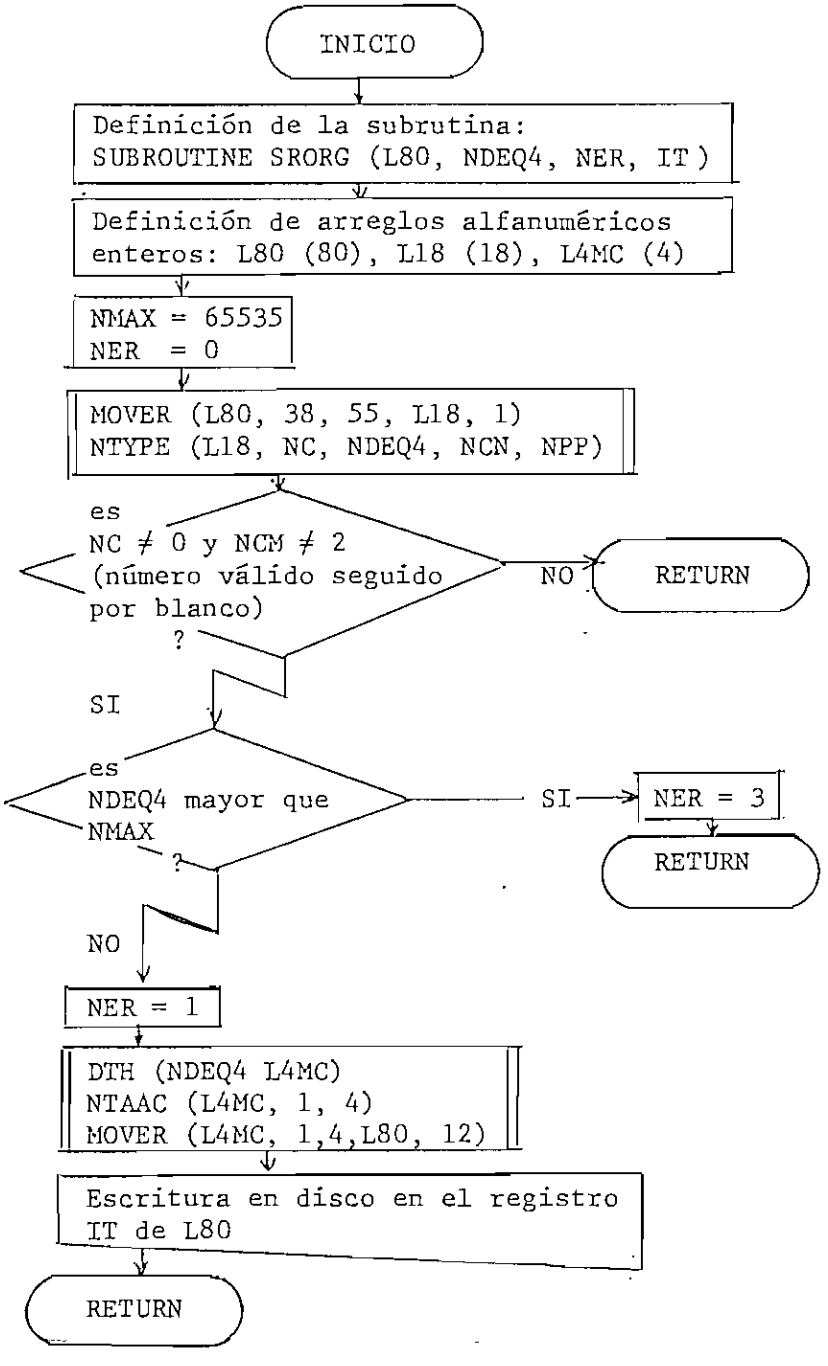
SUBRUTINA SRORG

Definición	: SUBROUTINE SRORG (L80, NDEQ4, NER, IT)
Propósito	: Subrutina que determina cuándo se tiene una pseudo-operación ORG.
Subprogramas llamados	: MOVER, NTYPE, DTH, NTACC, PERR.
Forma de utilización	: CALL SUBROUTINE SRORG (L80, NDEQ4, NER, IT)
Parámetros	: L80, NDEQ4, NER, IT
	L80 : Registro de entrada (instrucción) de 80 posiciones
	NDEQ4 : Equivalente decimal proveniente de la subrutina NTYPE.
	NER - 0 número de ORG no es válido
	- 3 excede capacidad de memoria
	IT : número de la instrucción analizada
Datos de entrada	: L80, NDEQ4, IT.
Datos de salida	: NER, L80.
Diagrama de flujo	: Fig. 4.28
Listado	: Ver Apéndice B, pág. 250

Funciones que realiza dentro del compilador:

Este subprograma encuentra si una pseudo-operación válida es ORG o no lo es. De acuerdo a ésto, se puede afirmar que la subrutina SRORG es una parte constitutiva del analizador semántico.

Fig. 4 .28 SUBRUTINA SRORG



SUBRUTINA SRRMB

Definición : SUBROUTINE SRRMB (L80, NQ4, NER, IT, NSH)

Propósito : Subrutina que identifica la pseudo-operación RMB.

Subprogramas llamados : MOVER, NTYPE, PERR, LABEL, DTH,NTAAC.

Forma de utilización : CALL SRRMB (L80, NQ4, NER, IT, NSH)

Parámetros : L80, NQ4, NER, IT, NSH

L80: instrucción de entrada de 80 posiciones.

NQ4: número decimal, contador de memoria que apunta a la etiqueta.

NER - 0 operando no número

- 1 pseudo-operación es RMB

- 2 no es etiqueta

- 3 (número + NQ4) mayor que 65535

IT : Entrega número de instrucción del programa

NSM - 1 primera pasada del assembler

2 segunda pasada del assembler

Datos de entrada : L80, NQ4, IT

Datos de salida : NER, NQ4, L80

Diagrama de flujo : Fig. 4.29

Listado : Ver Apéndice B, pág. 252 .

Funciones que realiza dentro del compilador:

Especifica que una pseudo-operación válida es RMB. El análisis que realiza pertenece al bloque semántico.

Fig. 4 .29 SUBRUTINA SRRMB

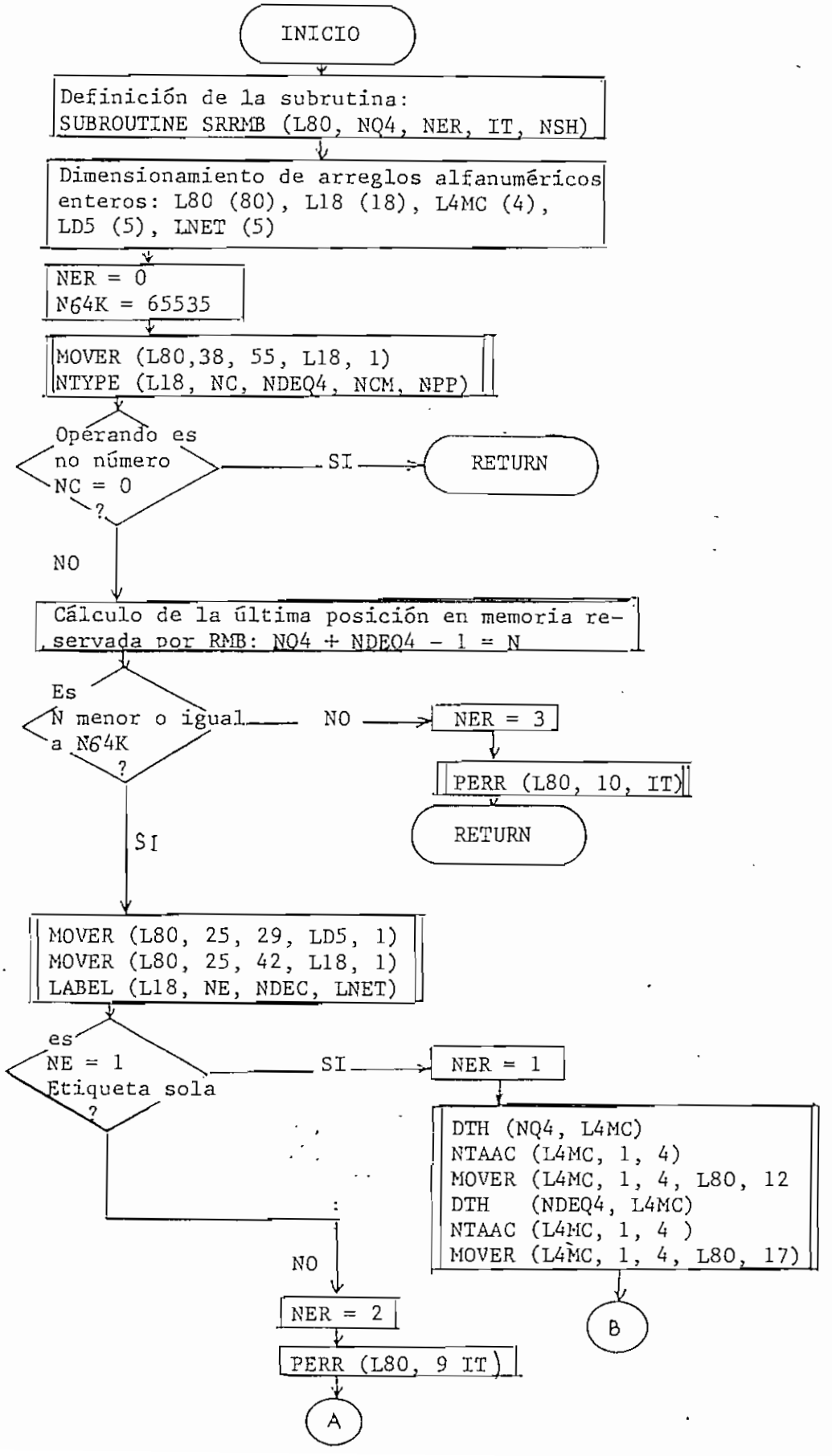
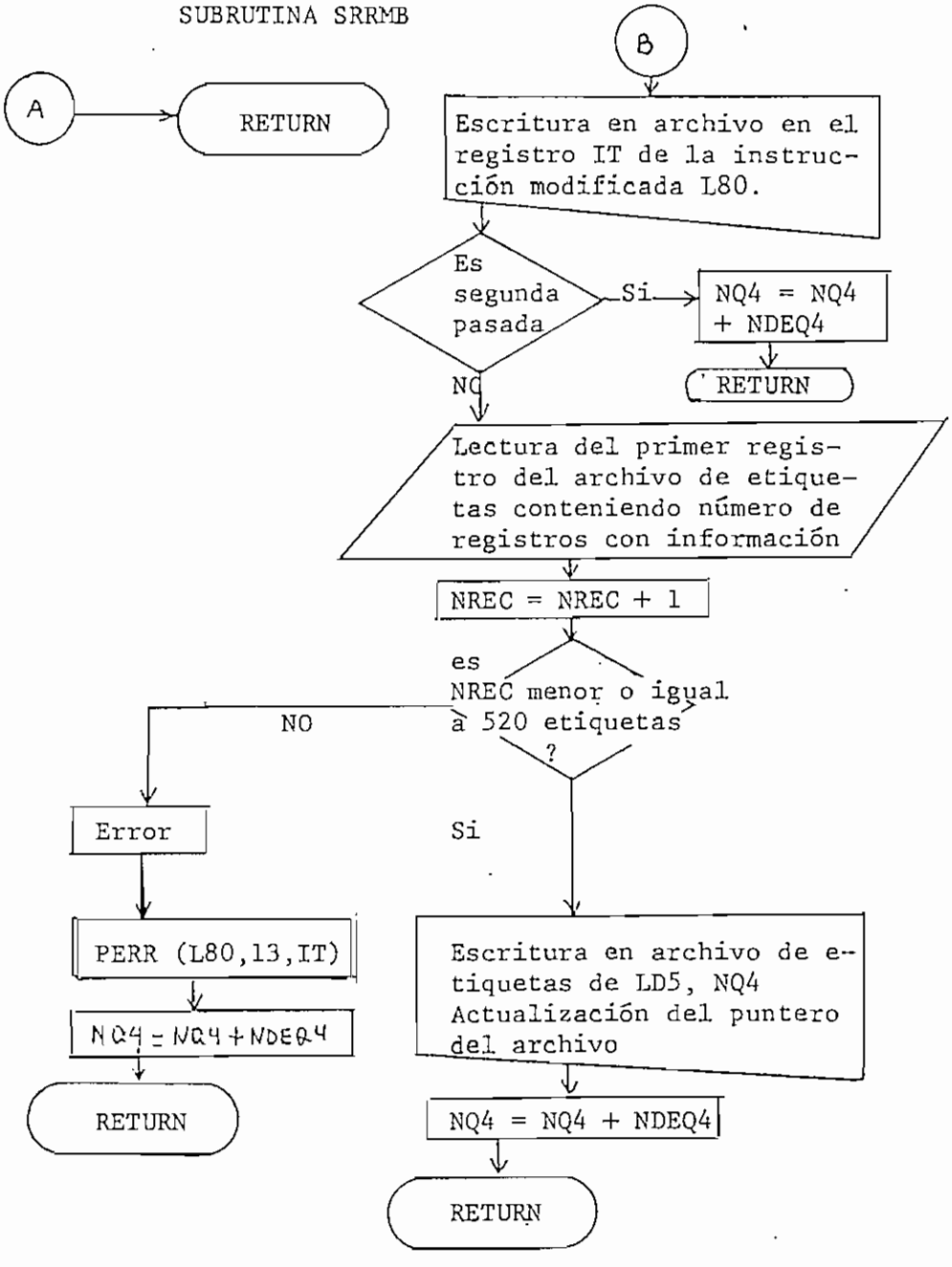


Fig. 4.29 SUBRUTINA SRRMB



SUBRUTINA SREQU

Definición	: SUBROUTINE SREQU (L80, NQ4, NER, IT, NSH)
Propósito	: Reconocimiento de pseudo-operación EQU
Subprogramas llamados	: MOVER, NTYPE, LABEL, DTH, NTAAC, PERR
Forma de utilización	: CAL SREQU (L80, NQ4, NER, IT, NSH)
Parámetros	: L80, NQ4, NER, IT, NSH

L80: arreglo de 80 posiciones que contiene una línea de programa.

NQ4: número decimal, contador de memoria que apunta a la etiqueta.

NER. - 0 operando no válido para pseudo operación

- 1 pseudo-operación es EQU

- 2 no tiene etiqueta

IT : número de línea de programa

NSH - 1 primera pasada del assembler

- 2 segunda pasada del assembler

Datos de entrada	: L80, NQ4, IT, NSH
Datos de salida	: L80, NQ4, NER
Diagrama de flujo	: Fig. 4.30
Listado	: Ver Apéndice B, pág.251 .

Funciones que realiza dentro del compilador:

Determina, de acuerdo a la forma en que se han colocado identificadores válidos, que la pseudo-operación es EQU o que no lo es. El análisis efectuado es semántico.

Fig. 4.30 SUBRUTINA SREQU

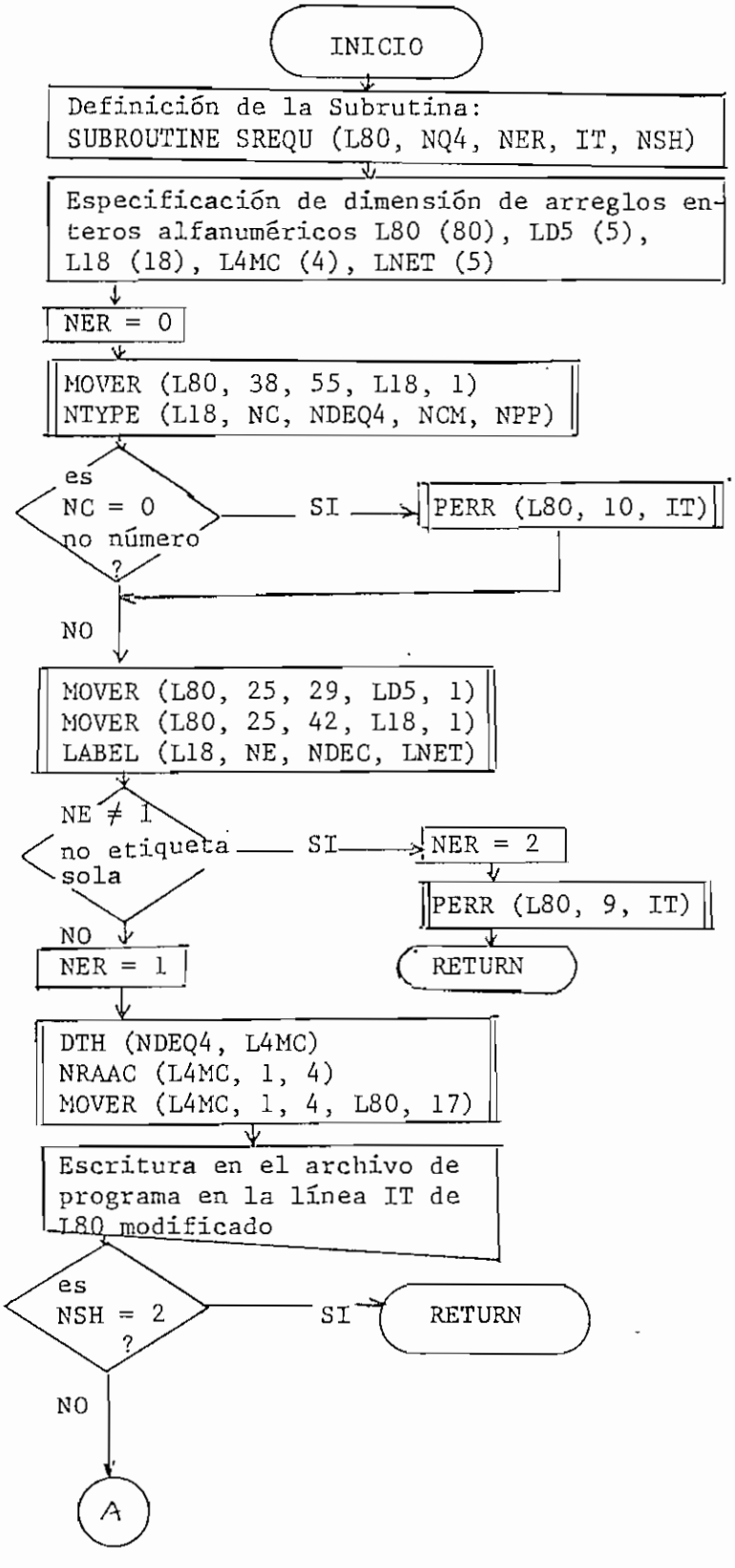
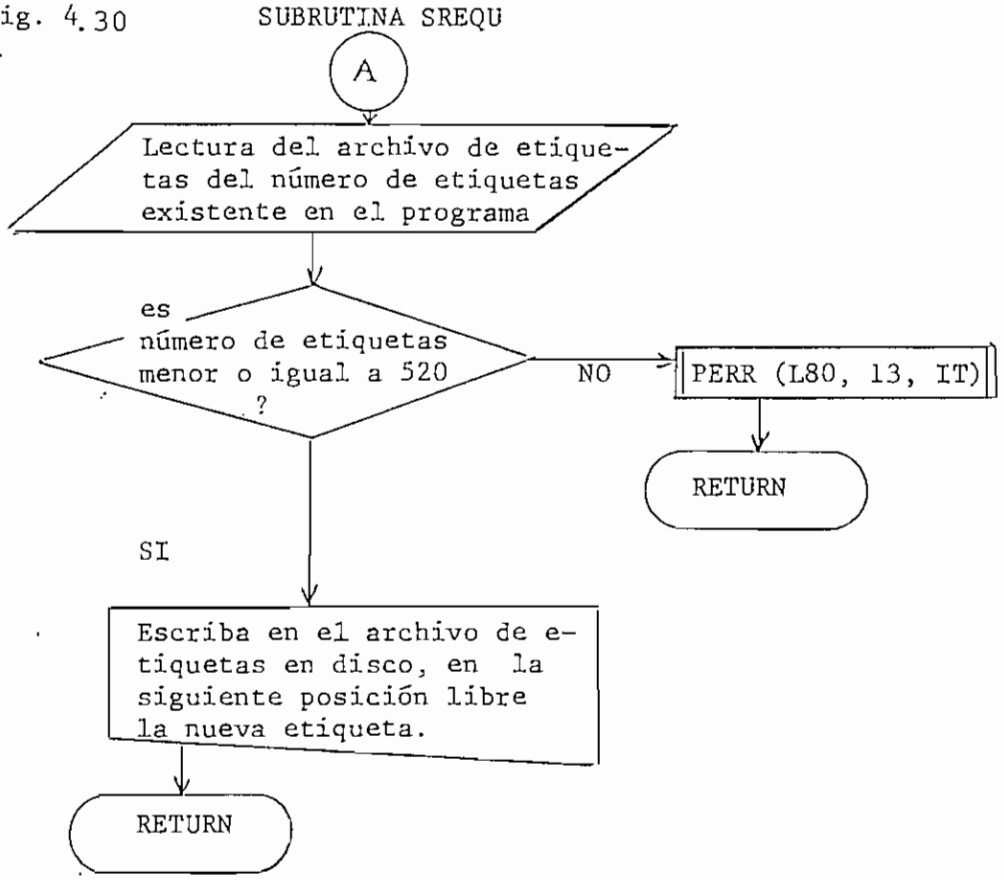


Fig. 4.30 SUBRUTINA SREQU



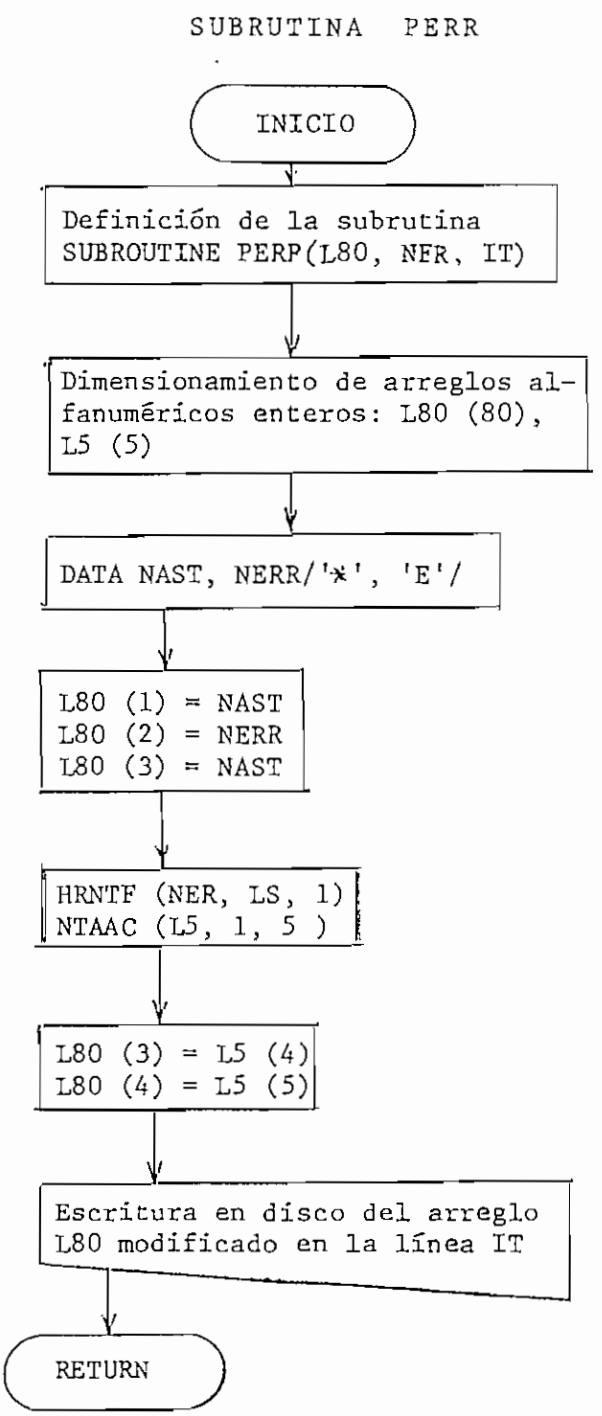
SUBRUTINA PERR

Definición	: SUBROUTINE PERR (L80, NER, IT)
Definición	: Escribe en las líneas de programas el código de error que corresponda.
Subprogramas llamados	: HRNTF y NTAAC
Forma de utilización	: CALL PERR (L80, NER, IT)
Parámetros	: L80, NER, IT
	L80: línea de entrada de 80 posiciones
	NER: número de error
	IT : número de línea del programa
Datos de entrada	: L80, NER, IT
Datos de salida	: L80 modificado
Diagrama de flujo	: Figura 4.31
Listado	: Ver Apéndice B, pág. 249.

Funciones que desempeña dentro del compilador:

Tiene que ver con el manejo de errores y la generación de código.

Fig. 4.31


SUBRUTINA PERR
INICIO
Definición de la subrutina
SUBROUTINE PERP(L80, NER, IT)
Dimensionamiento de arreglos al-
fanuméricos enteros: L80 (80),
L5 (5)
DATA NAST, NERR/'*', 'E'/
L80 (1) = NAST
L80 (2) = NERR
L80 (3) = NAST
HRNTF (NER, LS, 1)
NTAAC (L5, 1, 5)
L80 (3) = L5 (4)
L80 (4) = L5 (5)
Escritura en disco del arreglo
L80 modificado en la línea IT
RETURN

SUBRUTINA SRFCC

Definición	: SUBROUTINE SRFCC (L80, NQ4, NER, IT,NSH)
Propósito	: Reconoce y da el tratamiento necesario a una línea de programa que contenga la pseudo-operación FCC.
Subprogramas llamados	: DTH, NTAAC, MOVER, LABEL, PERR
Forma de utilización	: CALL SRFCC (L80, NQ4, NER, IT, NSH)
Parámetros	: L80, NQ4, NER, IT, NSH

L80: línea de entrada de 80 posiciones del programa.

NQ4: número decimal contador de memoria que apunta a la etiqueta.

NER: 0 - operando no número
1 - pseudo-opración es FCC
2 - no es etiqueta
3 - (número + etiqueta) mayor 65535

IT : número de instrucción del programa que entrega el programa principal.

NSH: número de pasada del assembler (1 ó 2)

Datos de entrada	: L80, NQ4, IT, NSH
Datos de salida	: L80, NQ4, NER
Diagrama de flujo	: Fig. 4.32
Listado	: Ver Apéndice B, pág.253.

Funciones que desempeña dentro del compilador:

Realiza una parte del análisis semántico, pues analiza si se tiene una pseudo-operación FCC válida. Colabora en la generación de código, en lo que concierne a esa directiva, escribiendo el contador hexadecimal de memoria.

Fig. 4.32 SUBRUTINA SRFCC

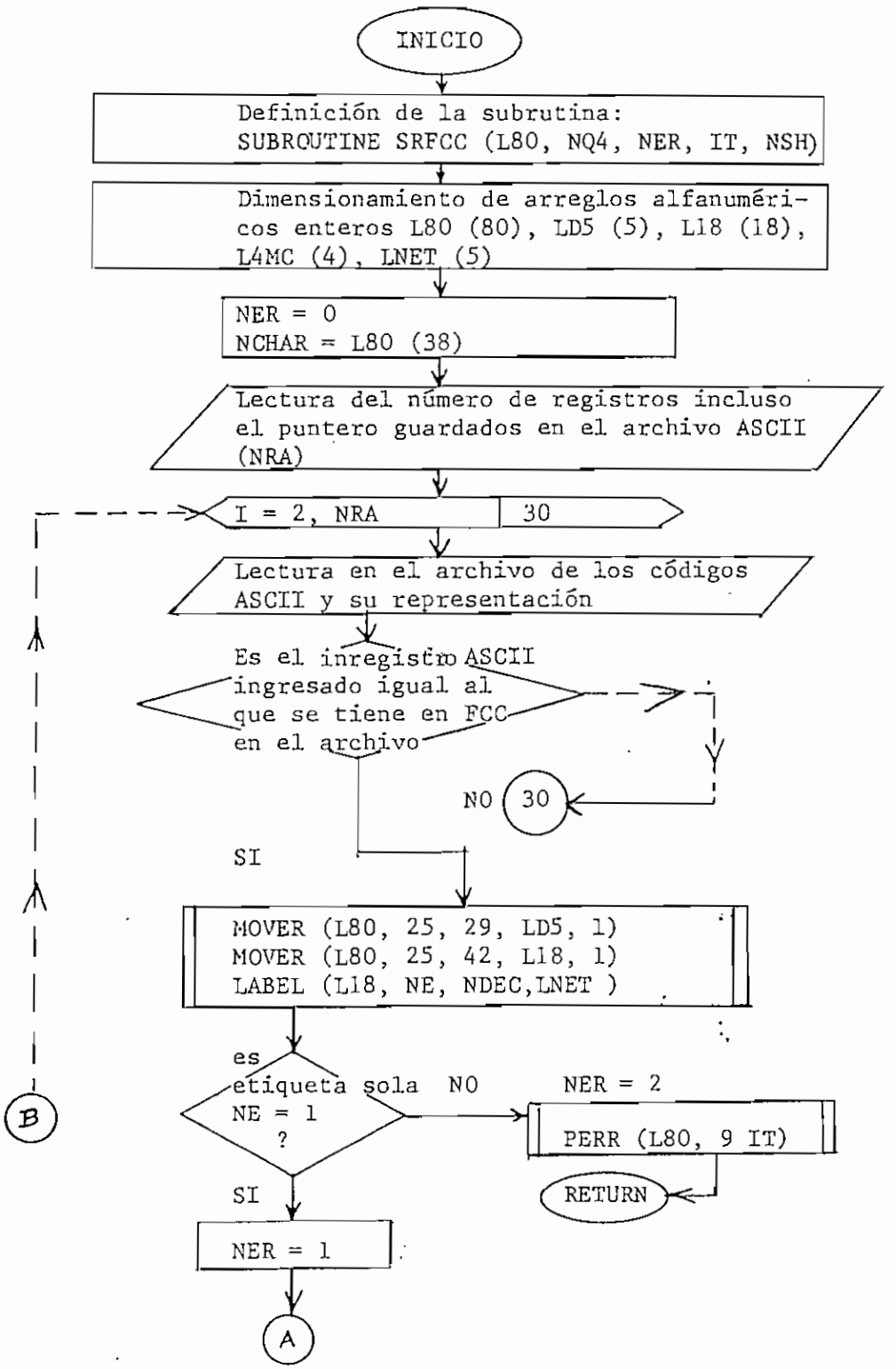
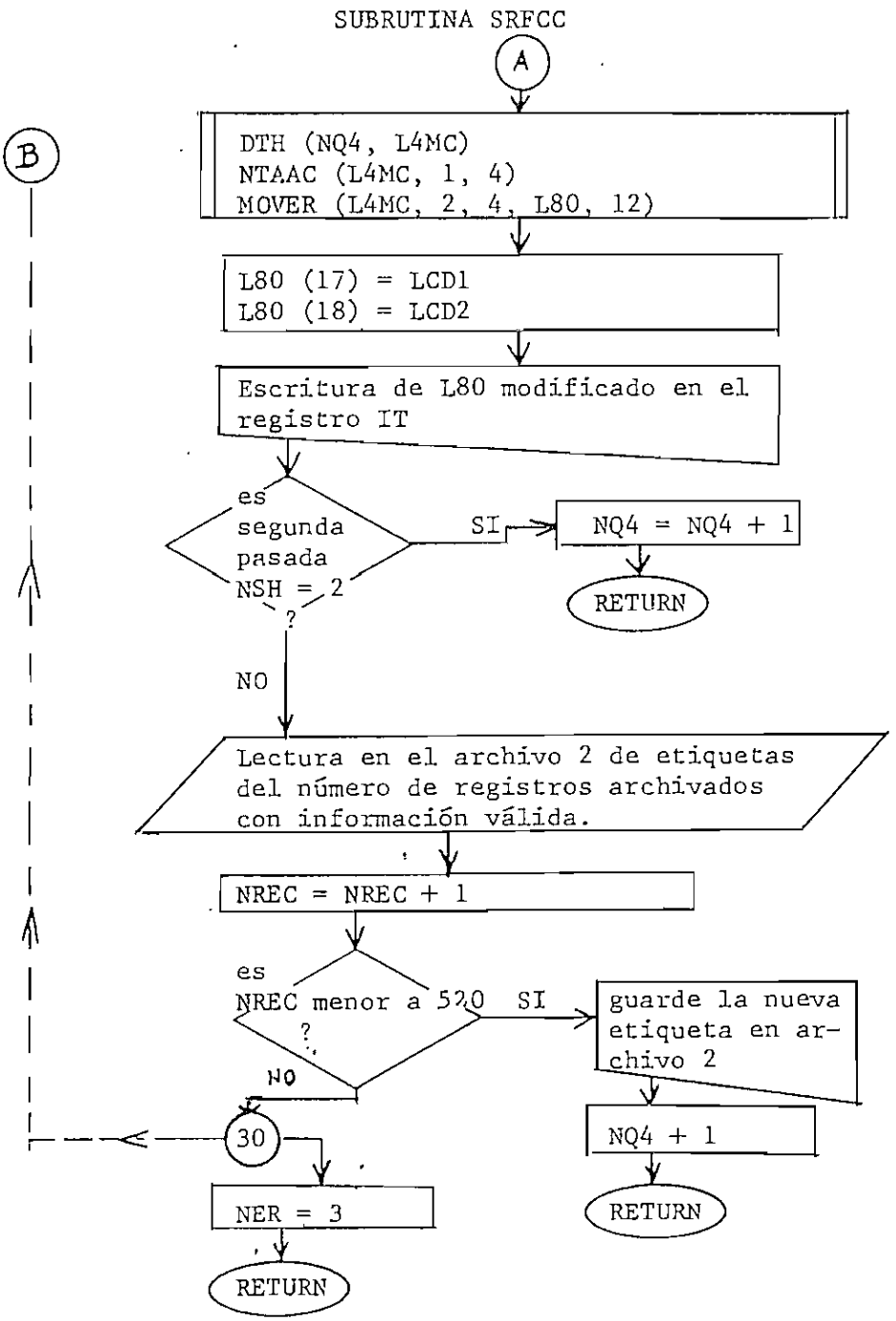


Fig. 4.32


SUBRUTINA SRFCC
A
B
DTH (NQ4, L4MC)
NTAAC (L4MC, 1, 4)
MOVER (L4MC, 2, 4, L80, 12)
L80 (17) = LCD1
L80 (18) = LCD2
Escritura de L80 modificado en el registro IT
es segunda pasada NSH = 2 ?
SI
NQ4 = NQ4 + 1
RETURN
NO
Lectura en el archivo 2 de etiquetas del número de registros archivados con información válida.
NREC = NREC + 1
es NREC menor a 520 ?
SI
guarde la nueva etiqueta en archivo 2
NQ4 + 1
RETURN
NO
30
NER = 3
RETURN

SUBRUTINA SRFCB

Definición : SUBROUTINE SRFCB (L80, NQ4, NER, IT,NSH)

Propósito : Reconoce y da un tratamiento adecuado a una línea de programa que contenga una pseudo-operación FCB.

Subprogramas llamados : MOVER, NTYPE, LABEL, DTH, NTAAC, PERR.

Forma de utilización : CALL SRFCB (L80), NQ4, NER, IT, NSH)

Parámetros : L80, NQ4, NER, IT, NSH

L80: líneas de entrada de 80 posiciones del programa

NQ4: número decimal contador de memoria que apunta a la etiqueta.

NER: 0 – operando no número
1 – pseudo-operación es FCB
2 – no es etiqueta
3 – (número + etiqueta) mayor que 65535.

Datos de entrada : L80, NQ4, IT, NSH

Datos de salida : L80, NER, NQ4.

Diagrama de flujo : Fig. 4.33

Listado : Ver Apéndice B, pág.254.

Funciones que desempeña dentro del compilador:

Realiza análisis semántico, pues reconoce si la pseudo operación es una FCB válida. En la generación de código toma parte escribiendo en el archivo fuente el contador hexadecimal de memoria para una directiva FCB sin errores.

Fig. 4.33

SUBRUTINA SRFCB

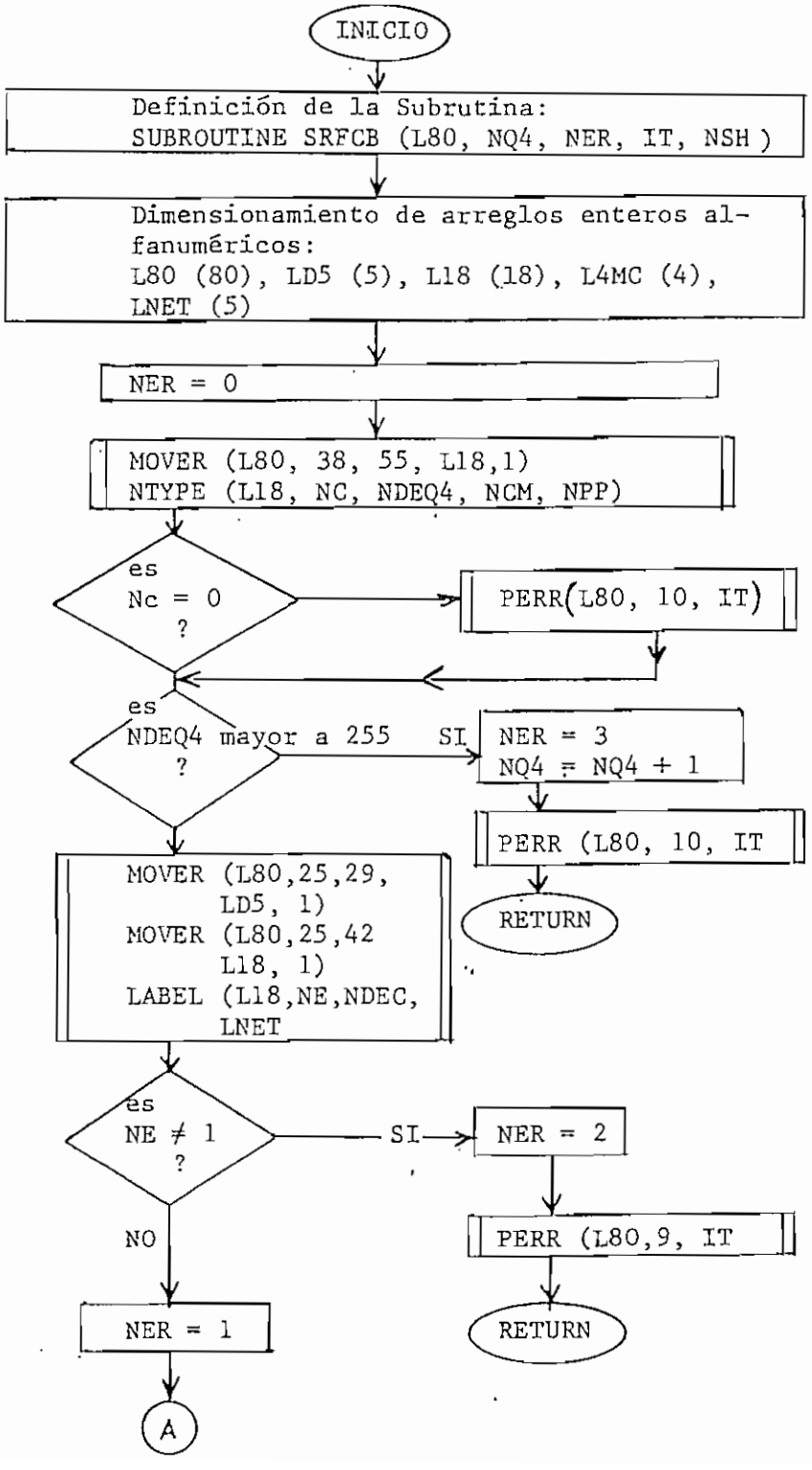
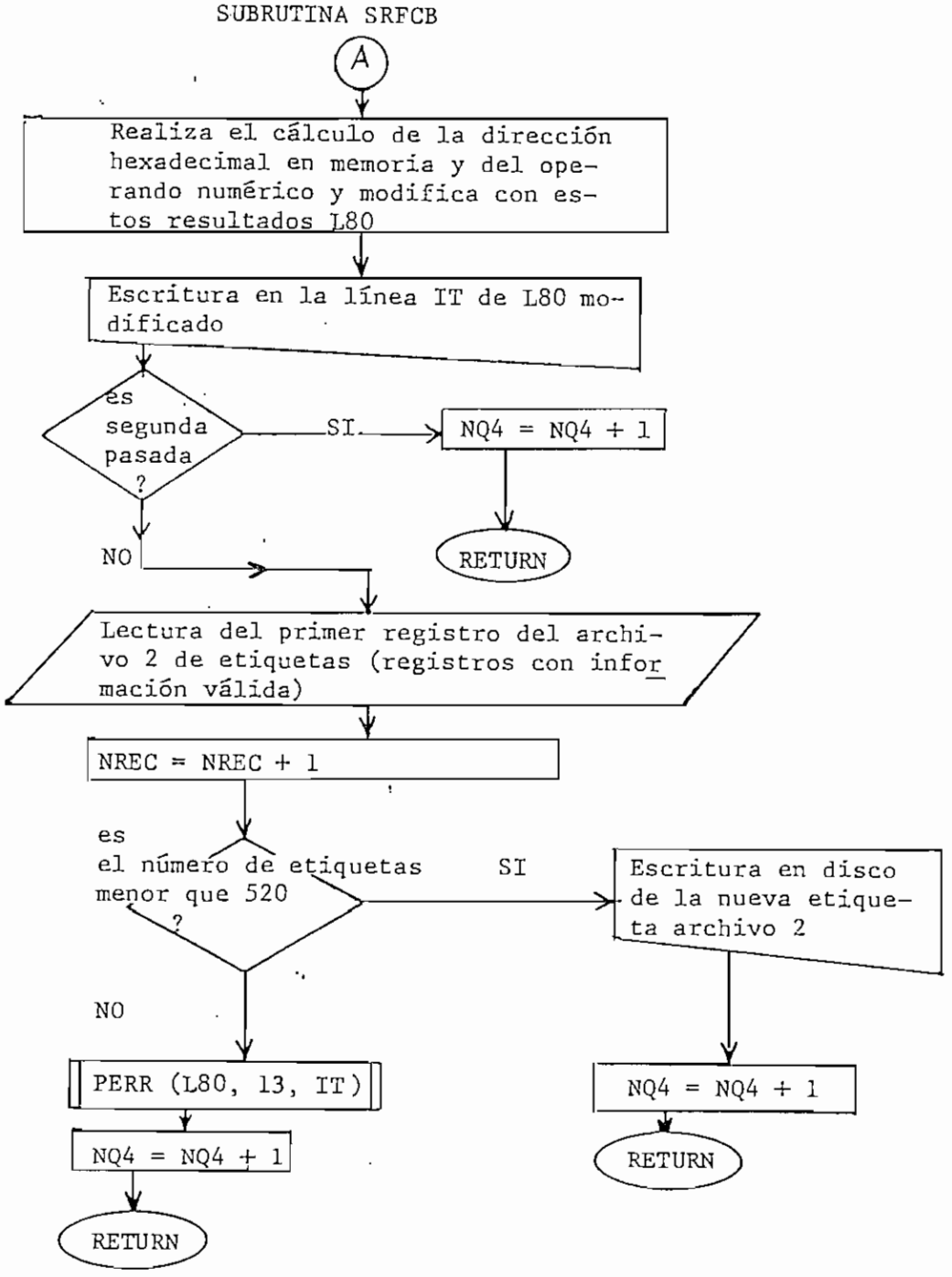


Fig. 4.33


SUBRUTINA SRFCB
A
Realiza el cálculo de la dirección hexadecimal en memoria y del operando numérico y modifica con estos resultados L80
Escritura en la línea IT de L80 modificado
es segunda pasada ?
SI
NQ4 = NQ4 + 1
RETURN
NO
Lectura del primer registro del archivo 2 de etiquetas (registros con información válida)
NREC = NREC + 1
es el número de etiquetas menor que 520 ?
SI
Escritura en disco de la nueva etiqueta archivo 2
NO
PERR (L80, 13, IT)
NQ4 = NQ4 + 1
RETURN
NQ4 = NQ4 + 1
RETURN

SUBRUTINA SCHLB

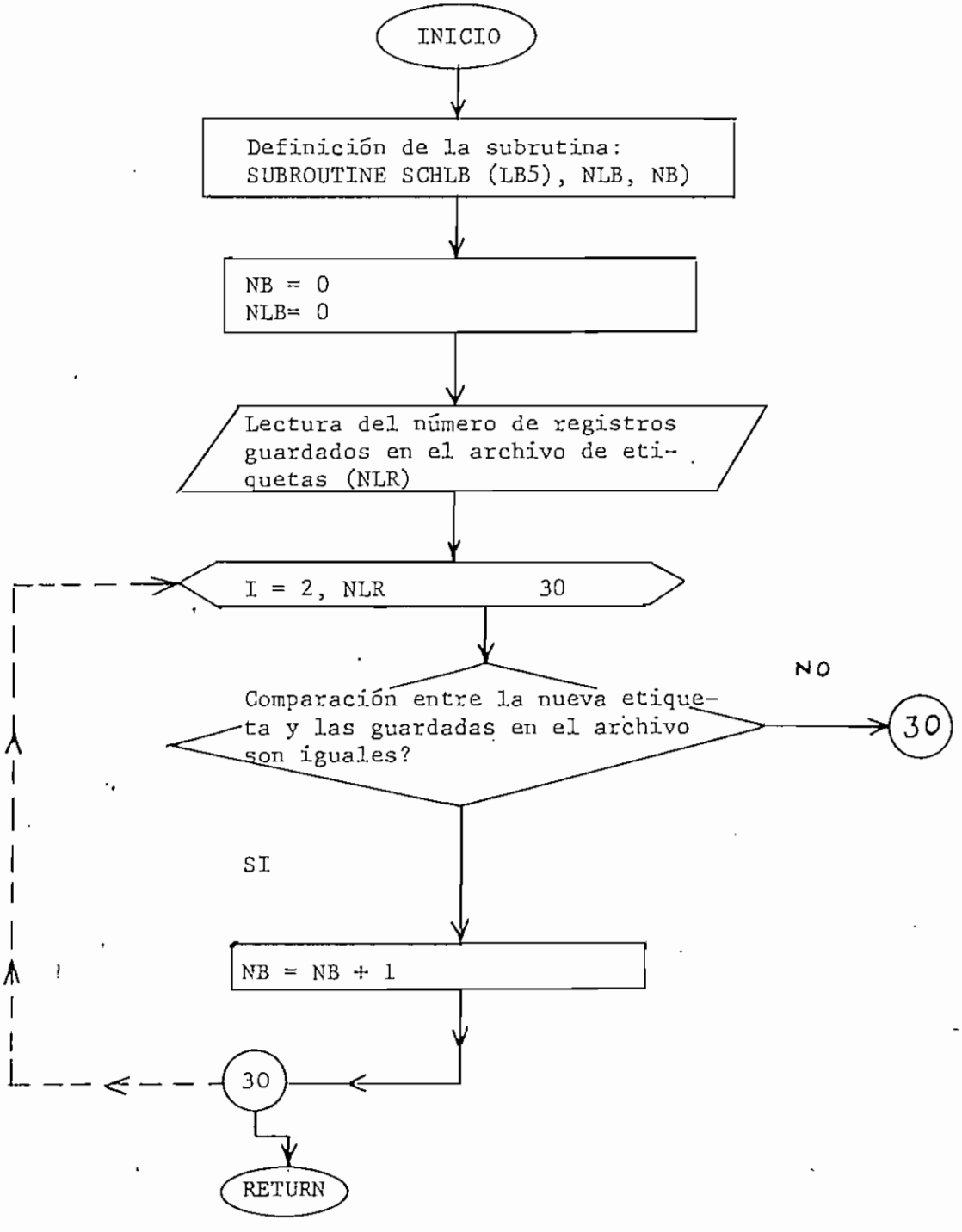
Definición	: SUBROUTINE SCHLB (LB5, NLB, NB)
Propósito	: Busca si existe una etiqueta ó un símbolo en la tabla de símbolos, las veces que está definida y la posición de memoria asociada a la etiqueta.
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL SCHLB (LB5 , NLB, NB)
Parámetros	: LB5, NLB, NB
	LB5: arreglo que contiene a la etiqueta
	NLB: arreglo que contiene a las etiquetas en el archivo 2
	NB : veces que la etiqueta está definida
Datos de entrada	: LB5
Datos de salida	: NLB, NB
Diagramas de flujo	: Fig. 4.34
Listado	: Ver Apéndice B, pág. 278.

Funciones que desempeña dentro del compilador:

Sirve de enlace entre el analizador sintáctico y la tabla de etiquetas.

Fig. 4.34

SUBRUTINA SCHLB



SUBRUTINA LCAR

Definición : SUBROUTINE LCAR (LCODE, LR, NX)

Propósito : Carga el código hexadecimal en el arreglo LCODE, según el tipo de direccionamiento.

Subprogramas llamados : Ninguno

Forma de utilización : CALL LCAR (LCODE, LR, NX)

Parámetros : LCODE, LR, NX

LCODE: arreglo en que se guarda el código NMOTECNICO que corresponda al tipo de direccionamiento

LR : arreglo de 16 elementos que contiene el código nmotécnico y sus diferentes hexadecimales.

NX : - 1 direccionamiento inmediato
- 2 direccionamiento directo
- 3 direccionamiento indexado
- 4 direccionamiento extendido
- 5 direccionamiento implícito
- 6 direccionamiento relativo

Datos de entrada : NX, LR

Datos de salida : LCODE

Diagrama de flujo : Fig. 4.35

Listado : Ver Apéndice B, pág.268.

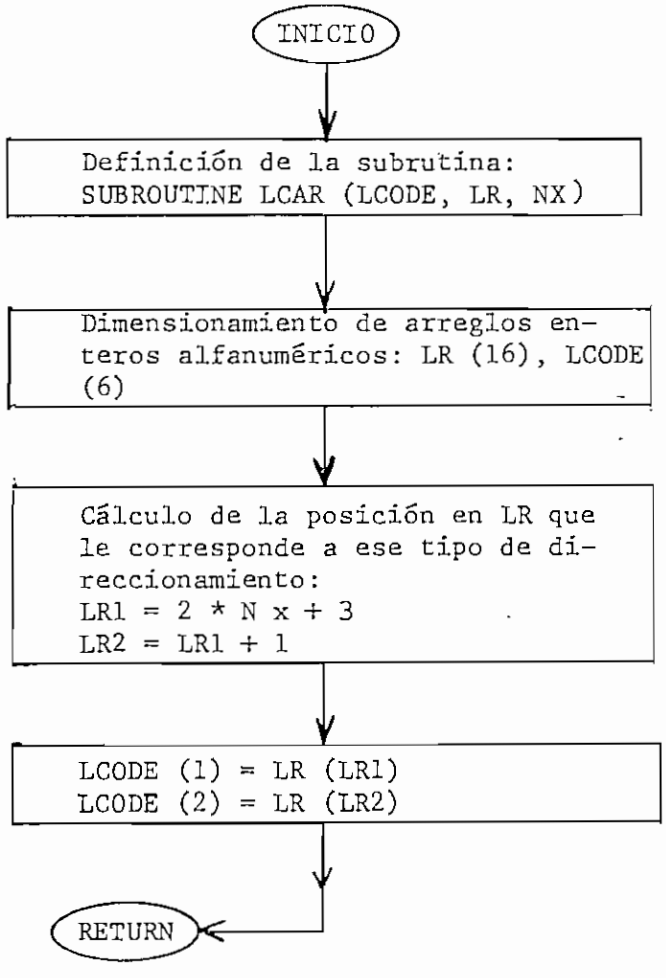
Funciones que desempeña dentro del compilador:

Determina la posición en que se encuentra en LR el código hexadecimal que corresponde a un tipo de direccionamiento. El programa que lo llama saca del archivo NMONI el registro LR y esta subrutina encuentra en él, al código hexadecimal del operador que va a ser utilizado luego, en la generación de código, es decir,

en la modificación del registro fuente L80.

Fig. 4.35

SUBRUTINA LCAR



SUBRUTINA COD1

Definición : SUBROUTINE COD1 (L80, NQ4, IT)

Propósito : Subrutina que genera el código del operando y la dirección hexadecimal correspondiente para el direccionamiento inmediato.

Subprogramas llamados : MOVER, CMP, NTYPE, PERR, DTH, NTACC

Forma de utilización : CALL COD1 (L80, NQ4, IT)

Parámetros : L80, NQ4, IT

L80: línea de entrada de 80 posiciones del programa

NQ4: contador de la posición en memoria, número decimal

IT : número de línea de programa

Datos de entrada : L80, NQ4, IT

Datos de salida : L80, NQ4

Diagrama de flujo : Fig. 4.36

Listado : Ver Apéndice B, pág.269.

Funciones que realiza dentro del compilador:

Este subprograma actúa en la generación de código.

Fig. 4.36

SUBRUTINA COD1

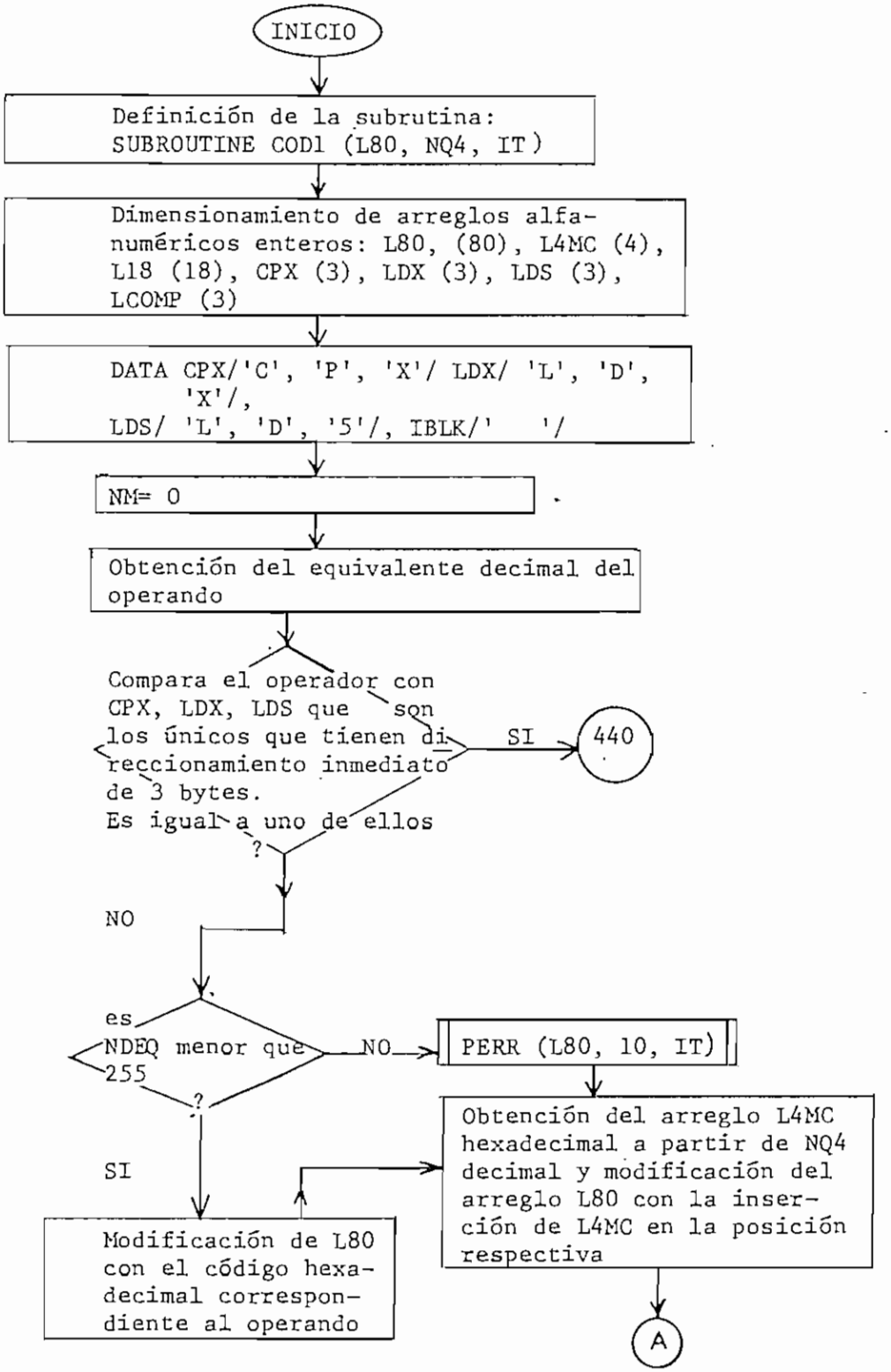
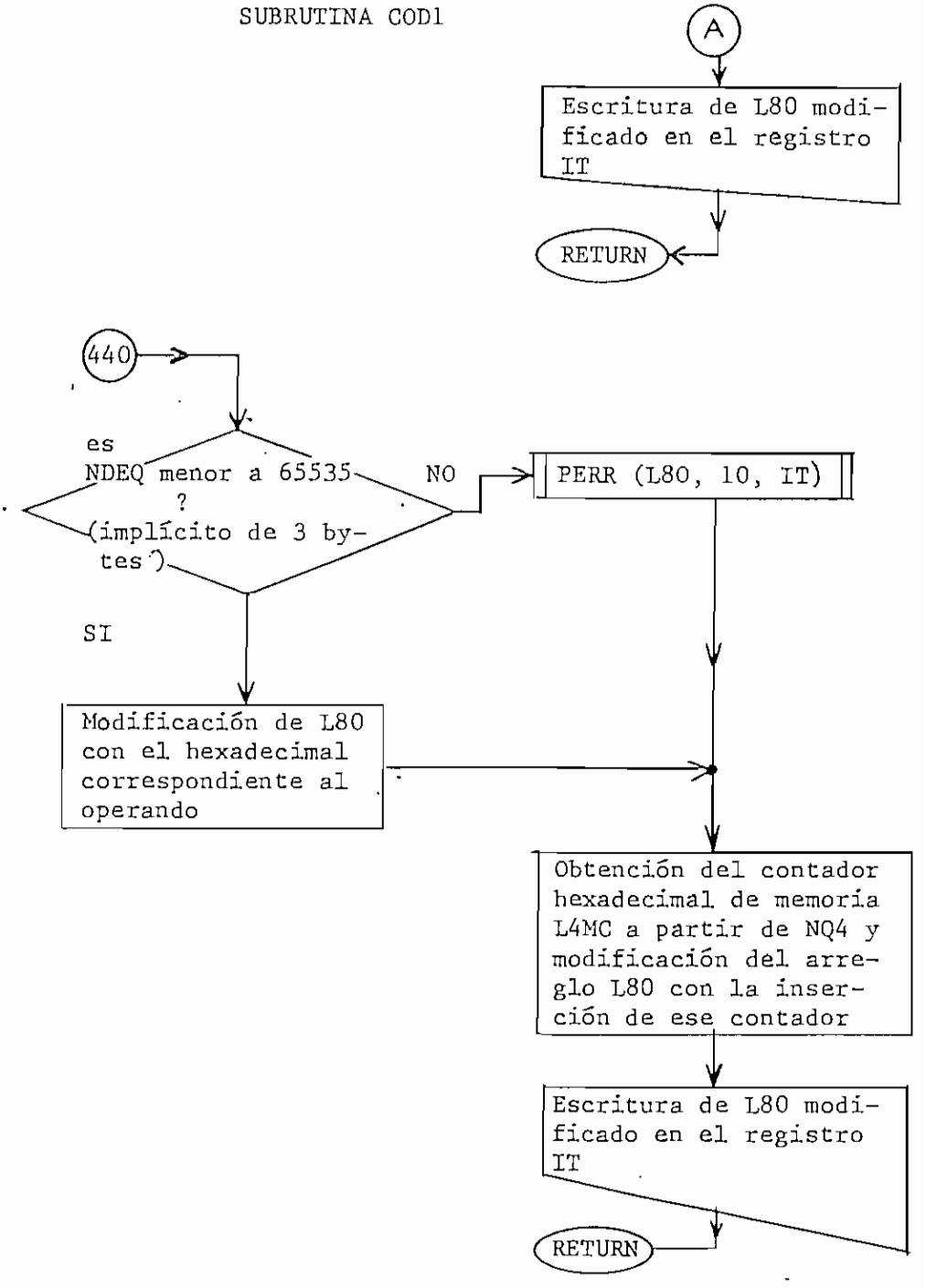


Fig. 4 .36


SUBRUTINA COD1
A
Escritura de L80 modificado en el registro IT
RETURN
440
es NDEQ menor a 65535 ? (implícito de 3 bytes)
NO
PERR (L80, 10, IT)
SI
Modificación de L80 con el hexadecimal correspondiente al operando
Obtención del contador hexadecimal de memoria L4MC a partir de NQ4 y modificación del arreglo L80 con la inserción de ese contador
Escritura de L80 modificado en el registro IT
RETURN

SUBRUTINA COD24

Definición	: SUBROUTINE COD24 (L80, NQ4, LCOD, NSH, IT)
Propósito	: Genera el código del operando y dirección en memoria para el caso de direccionamiento directo o extendido
Subprogramas llamados	: MOVER, CMP, LABEL, NTYPE, PERR, DTH, NTAAC, SCHLB.
Forma de utilización	: CALL COD24 (L80, NQ4, LCOD, NSH, IT)
Parámetros	: L80, NQ4, LCOD, NSH, IT
	L80: línea de 80 posiciones de entrada y salida ya modificada
	NQ4: contador de memoria (decimal)
	LCOD: Código hexadecimal del operador
	NSH: - 1 primera pasada del assembler - 2 segunda pasada del assembler
	IT : número de línea del programa
Datos de entrada	: L80, NQ4, NSH, IT, LCOD
Datos de salida	: L80, NQ4
Diagrama de flujo	: Fig. 4.37
Listado	: Ver Apéndice B, pág. 270.

Funciones que desempeña dentro del compilador:

Este subprograma forma parte del generador de código.

Fig. 4 .37

SUBRUTINA COD24

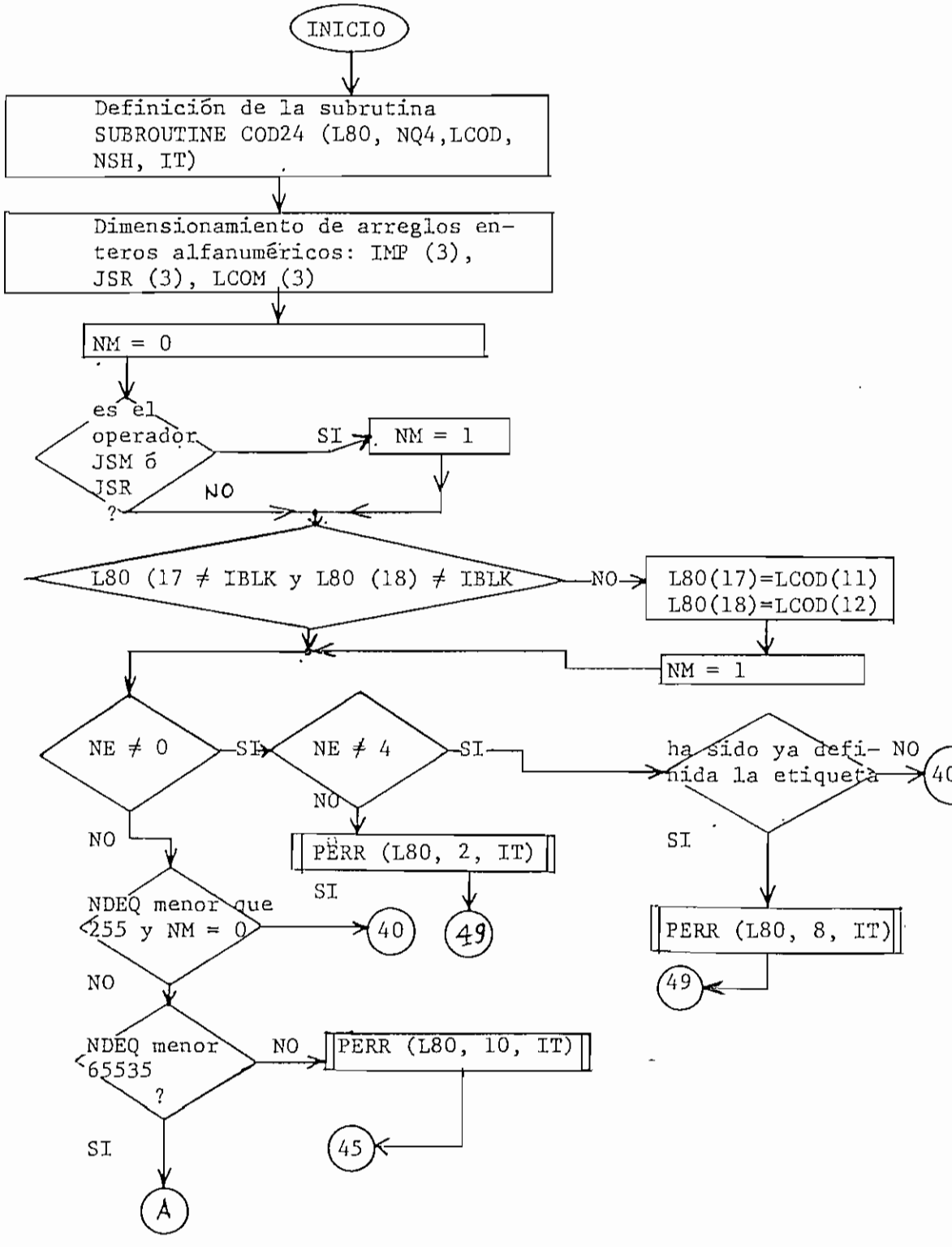


Fig. 4 .37

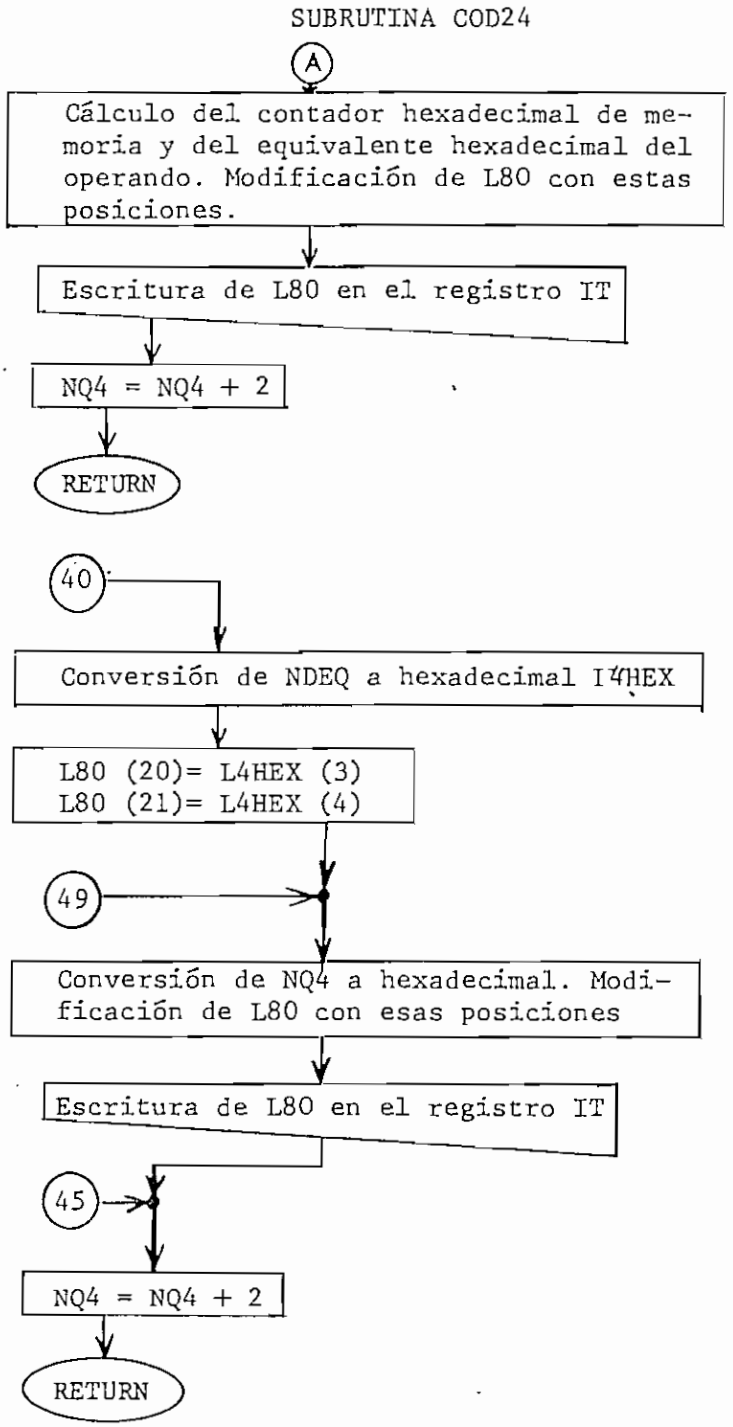
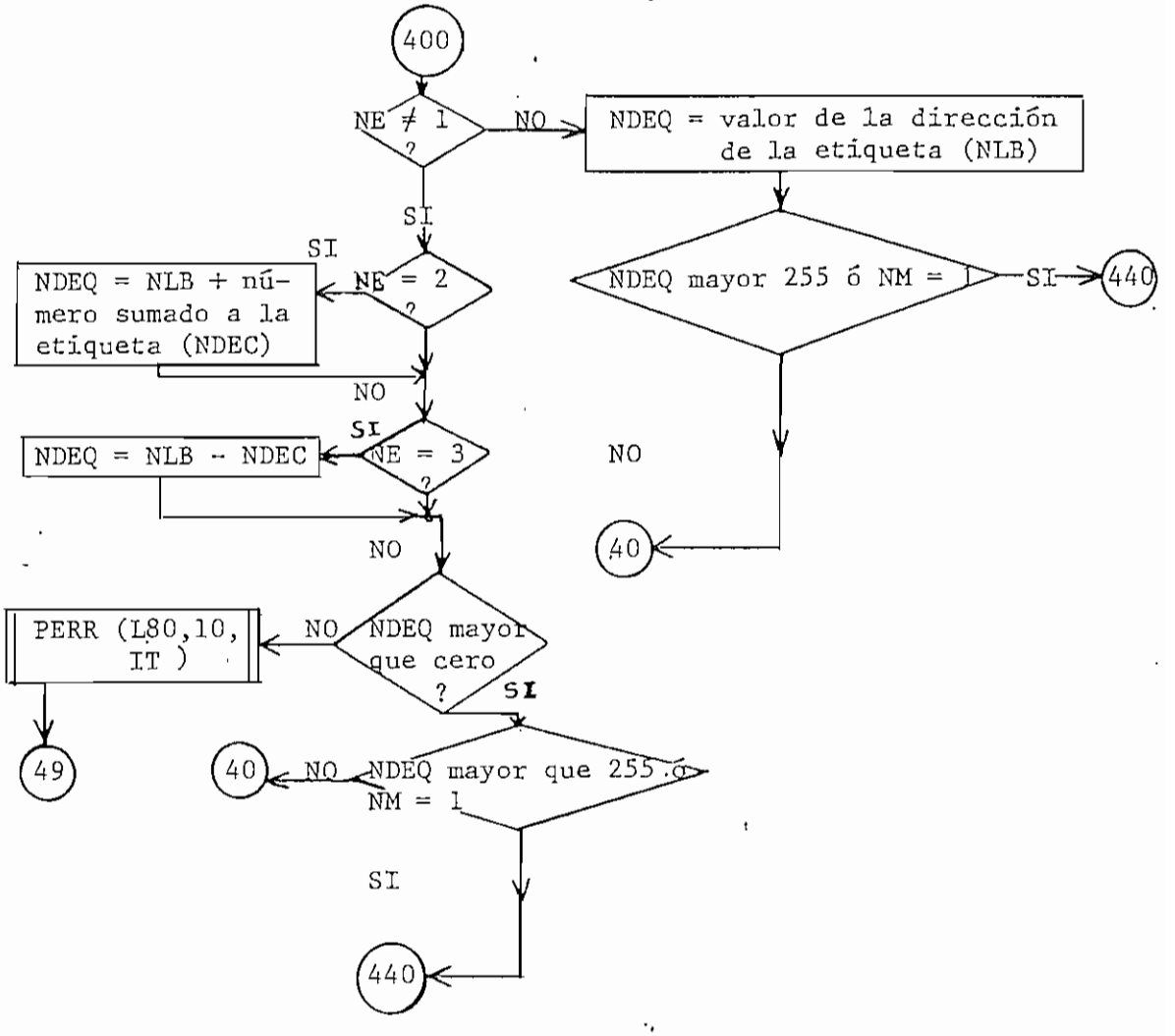

SUBRUTINA COD24
A
Cálculo del contador hexadecimal de memoria y del equivalente hexadecimal del operando. Modificación de L80 con estas posiciones.
Escritura de L80 en el registro IT
NQ4 = NQ4 + 2
RETURN
40
Conversión de NDEQ a hexadecimal I4HEX
L80 (20)= L4HEX (3)
L80 (21)= L4HEX (4)
49
Conversión de NQ4 a hexadecimal. Modificación de L80 con esas posiciones
Escritura de L80 en el registro IT
45
NQ4 = NQ4 + 2
RETURN

Fig. 4 .37

SUBRUTINA COD24



SUBRUTINA COD3

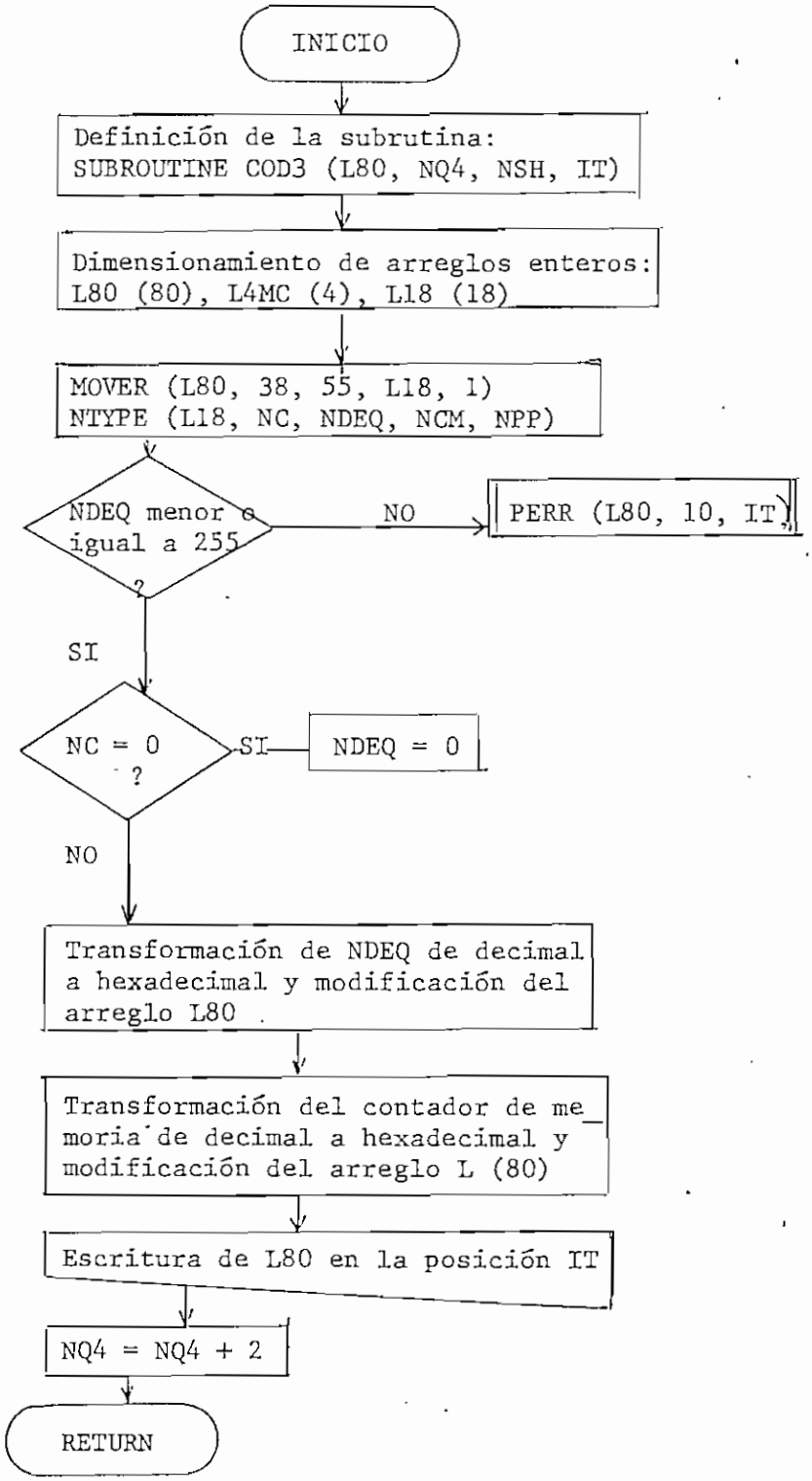
Definición	: SUBROUTINE COD3 (L80, NQ4, NSH, IT)
Propósito	: Genera el código del operando y dirección en memoria para el caso de direccionamiento inexado.
Subprogramas llamados	: MOVER, NTYPE, PERR, DTH, NTAAC.
Forma de utilización	: CALL COD3 (L80, NQ4, NSH, IT)
Parámetros	: L80, NQ4, NSH, IT
	L80: línea del programa de 80 posiciones
	NQ4: contador de memoria (decimal)
	NSH: - 1 primera pasada del assembler - 2 segunda pasada del assembler
	IT : número de línea del programa.
Datos de entrada	: L80, NQ4, NSH, IT
Datos de salida	: NQ4, L80 modificado
Diagrama de flujo	: Fig. 4.38
Listado	: Ver Apéndice B, pág. 272.

Funciones que desempeña dentro del compilador:

Realiza parte de la generación de código.

Fig. 4.38

SUBRUTINA COD3



SUBRUTINA COD5

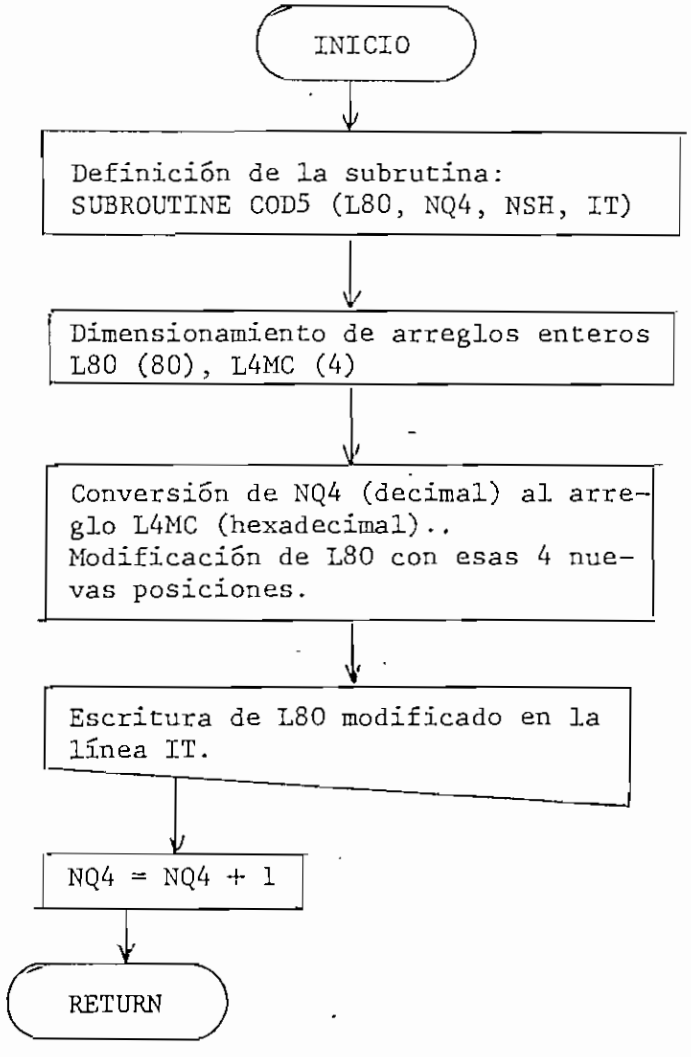
Definición	: SUBROUTINE COD5 (L80, NQ4, NSH, IT)
Propósito	: Generar las direcciones de memoria correspondientes a una línea del programa que contenga direccionamiento implícito.
Subprogramas llamados	: DTH, NTAAC, MOVER
Forma de utilización	: CALL COD5 (L80, NQ4, NSH, IT)
Parámetros	: L80, NQ4, NSH, IT
	L80: línea del programa de 80 posiciones
	NQ4: contador de memoria (decimal)
	NSH: - 1 primera pasada del assembler - 2 segunda pasada del assembler
	IT : - número de línea del programa
Datos de entrada	: L80, NQ4, NSH, IT
Datos de salida	: L80, NQ4
Diagrama de flujo	: Fig. 4.39
Listado	: Ver Apéndice B, pág. 273.

Funciones que desempeña dentro del compilador:

Forma parte del generador de código.

Fig. 4.39

SUBRUTINA COD5



SUBRUTINA COD6

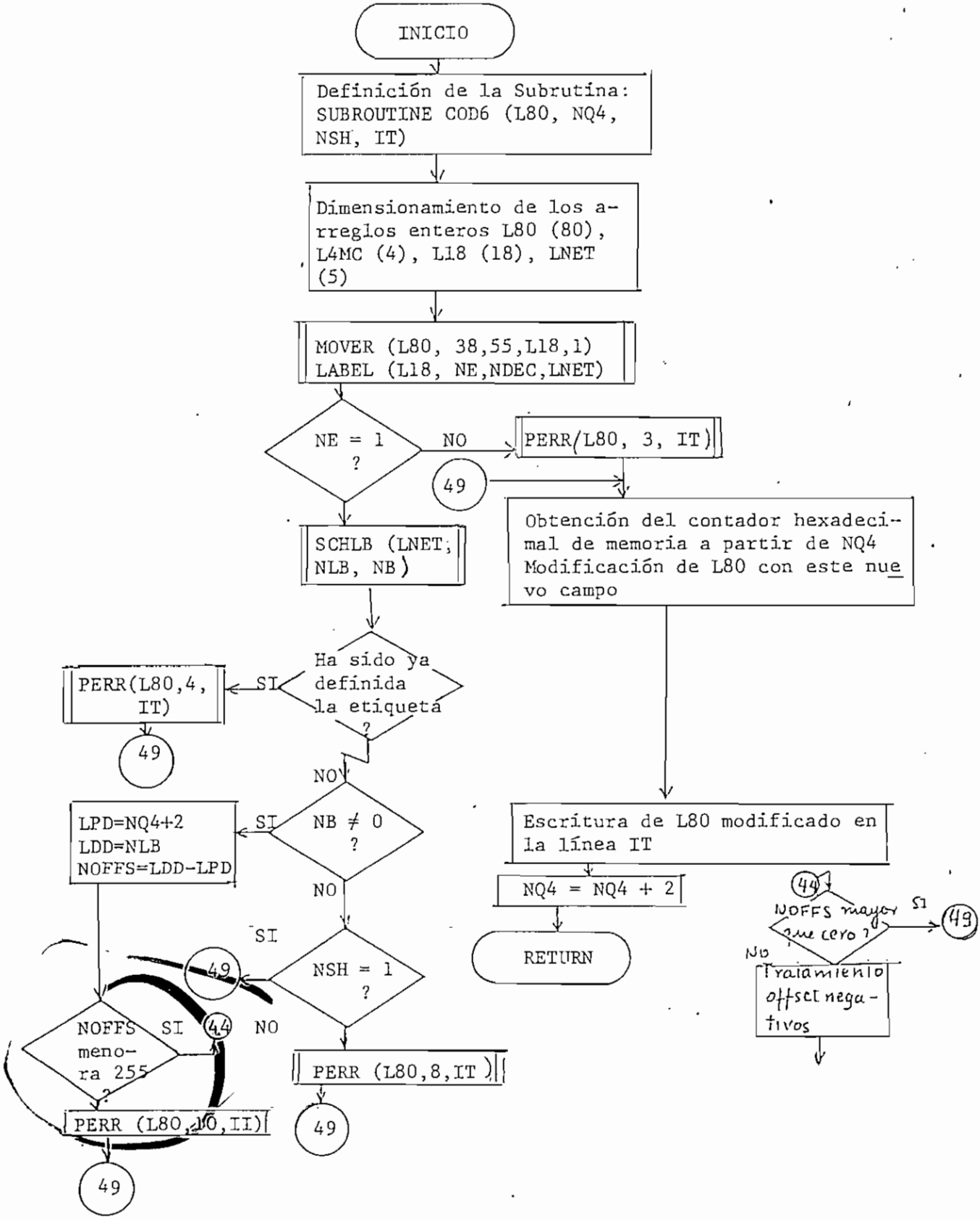
Definición	: SUBROUTINE COD6 (L80, NQ4, NSH, IT)
Propósito	: Generar el código del operando y la dirección en memoria para el caso de direccionamiento relativo.
Subprogramas llamados	: MOVER, LABEL, PERR, SCHLB, NTAAC.
Forma de utilización	: CALL COD6 (L80, NQ4, NSH, IT)
Parámetros	: L80, NQ4, NSH, IT
	L80: línea de programa de 80 posiciones
	NQ4: contador de memoria (decimal)
	NSH: - 1 primera pasada del assembler 2 segunda pasada del assembler
	IT : número de línea de programa
Datos de entrada	: L80, NQ4, NSH, IT
Datos de salida	: L80, NQ4
Diagrama de flujo	: Fig. 4.40
Listado	: Ver Apéndice B, pág. 274.

Funciones que desempeña en el compilador:

Realiza tareas del generador de código.

Fig. 4.40

SUBRUTINA COD6



SUBRUTINA CMP

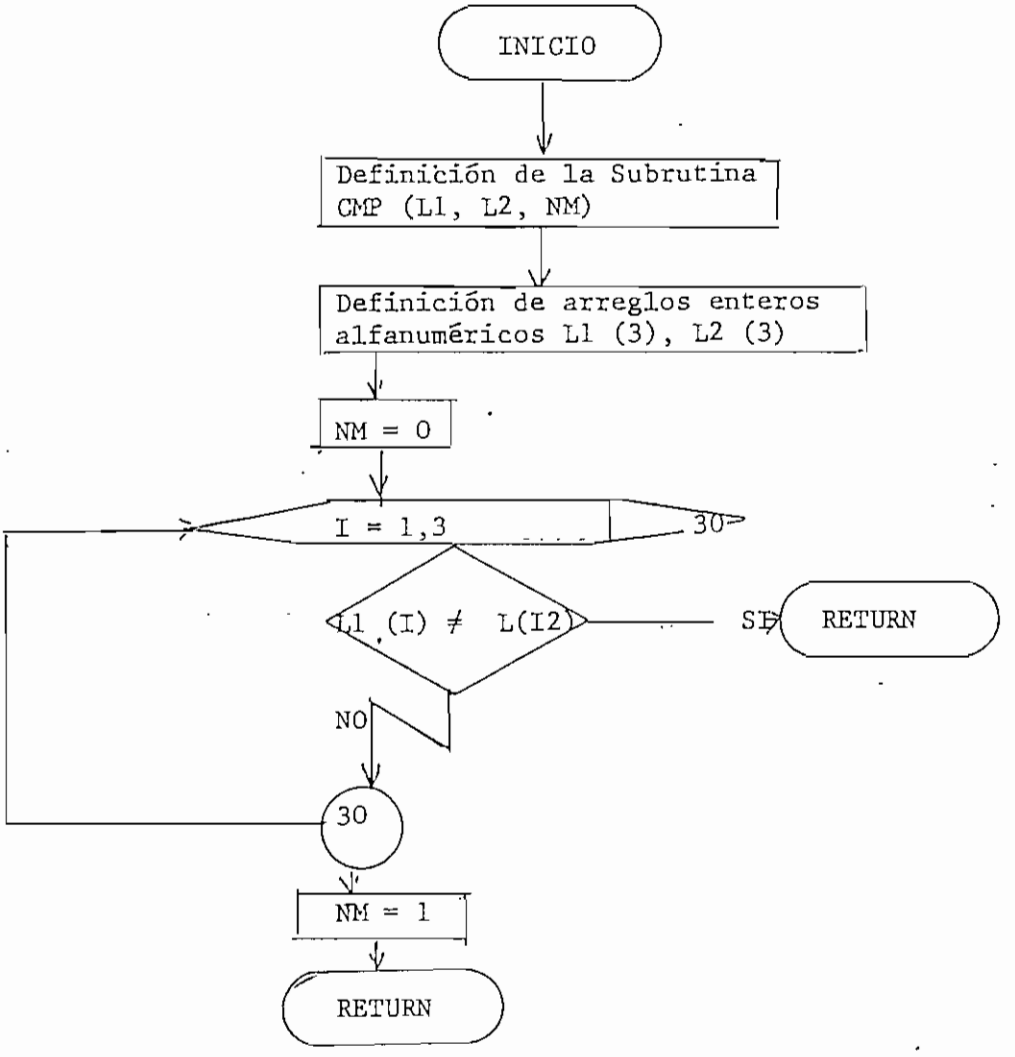
Definición	: SUBROUTINE CMP (L1, L2, NM)
Definición	: Subrutina que compara dos arreglos enteros alfanuméricos y determina si son iguales o si no lo son.
Subprogramas llamados	: Ninguno
Forma de utilización	: CALL CMP
Parámetros	: L1, L2, NM
	L1: arreglo entero alfanumérico 1
	L2: arreglo entero alfanumérico 2
	NM: - 0 no son iguales
	- 1 son iguales
Datos de entrada	: L1, L2
Datos de salida	: NM
Diagrama de flujo	: Fig. 4.41
Listado	: Ver Apéndice B, pág. 255

Funciones que realiza dentro del compilador:

Efectúa un análisis de si el operando que se tiene es igual a un operando válido o no. Por lo tanto realiza un análisis sintáctico.

Fig. 4
.41

SUBRUTINA CMP



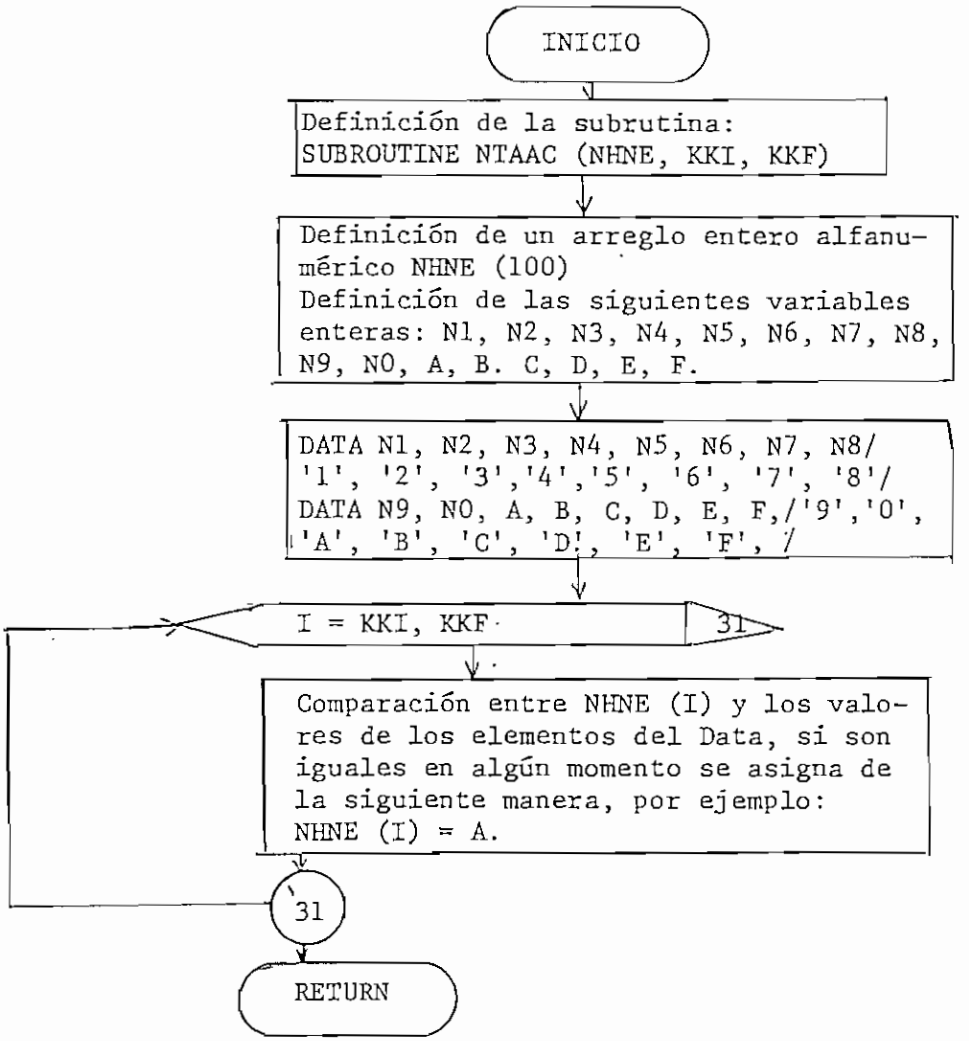
SUBRUTINA NTAAC

Definición	: NTAAC (NHNE, KKI, KKF)
Propósito	: Conversión de un arreglo numérico entero a un arreglo entero alfanumérico.
Subprogramas llamados	: ninguno
Forma de utilización	: CALL NTAAC (NHNE, KKI, KKF)
Parámetros	: NHNE, KKI, KKF
	NHNE: arreglo numérico a ser convertido o arreglo alfanumérico ya convertido
	KKI : posición inicial del arreglo a convertirse
	KKF : posición final del arreglo a convertirse
Datos de entrada	: NHNE, KKI, KKF
Datos de salida	: NHNE
Diagrama de flujo	: Fig. 4.42
Listado	: Ver Apéndice B, pág.276 .

Funciones que realiza dentro del compilador:

Se la utiliza para convertir arreglos decimales en hexadecimales. Al igual que la subrutina ATNAC, ésta es útil en la interrelación de las fases, obteniendo resultados intermedios que deben transmitirse de un proceso a otro durante la compilación.

Fig. 4.42 SUBRUTINA NTAAC



SUBRUTINA CGPR

Definición	: SUBROUTINE CGPR
Propósito	: Secuencializa las líneas de instrucción en assembler y objeto.
Subprogramas llamados	: ninguno
Forma de utilización	: CALL CGPR
Parámetros	: No tiene
Diagrama de flujo	: Fig. 4.43
Listado	: Ver Apéndice B, pág. 247.

Funciones que realiza dentro del compilador:

Realiza una parte dela generación de código.

Fig. 4.43

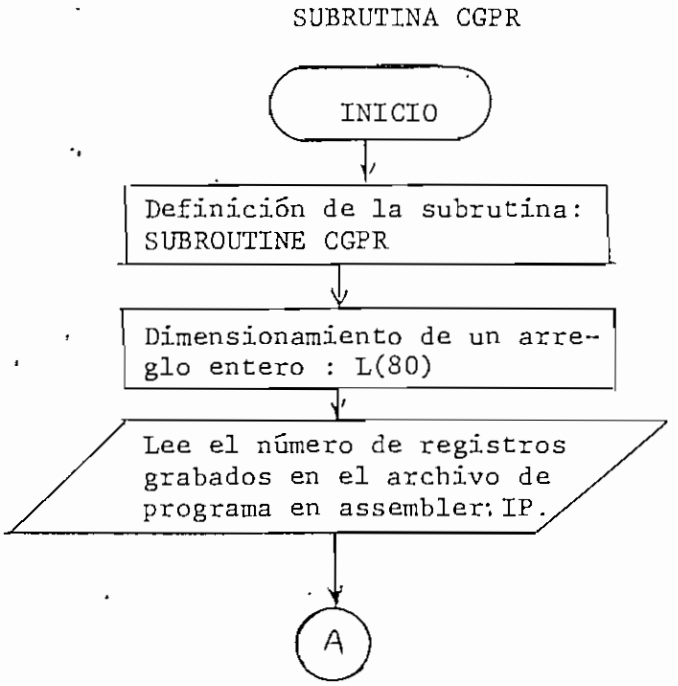
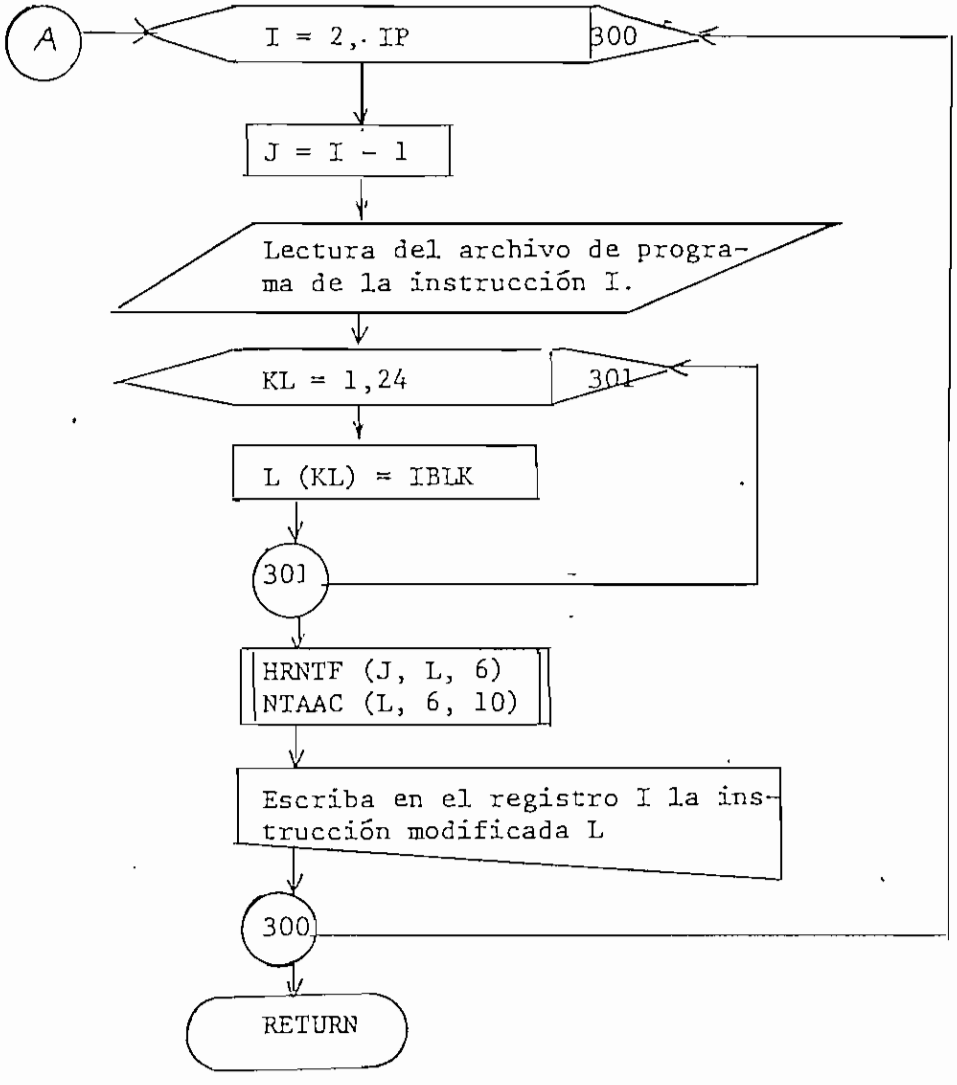


Fig. 4.43 SUBRUTINA CGPR



C A P I T U L O V

USO DE CROSS-ASSEMBLER DESARROLLADO Y LA ILUSTRACION DE SU UTILIZACION CON VARIOS EJEMPLOS.

5.1 INTRODUCCION.

En este Capítulo se van a ver algunos ejemplos de programas en los que se pueda apreciar los aspectos más importantes de la ejecución del ensamblador, el trabajo del editor en la carga de programas, la detección de errores, la impresión de mensajes de error, uso de subrutinas, saltos, lazos, pseudo-operaciones y ensamblajes satisfactoriamente realizados.

Se empezará por demostrar cómo funciona el editor en la carga de un programa y posteriormente se procederá a desarrollar programas que enfoquen distintas facetas del ensamblador, se mostrará su codificación en assembler y los resultados del proceso de compilación.

5.2 EJEMPLO DE CARGA DE UN PROGRAMA (edición)

Se va a programar una suma de 2 números de 16 bits, el primero ubicado en las localidades de memoria 0060 y 0061; y, el otro, en las localidades 0062 y 0063.

Los ocho bits más significativos de cada número, están en las localidades de memoria 0060 y 0062, respectivamente. Se guardará el resultado en las localidades de memoria 0064 y 0065 con los bits mayores en 0064. (Todas las direcciones son hexadecimales).

- Datos de un problema de ejemplo

(0060) = 52

(0061) = 1C

(0062) = 24

(0063) = CA

Resultado: 52IC + 24CA = 76E6

(0064) = 76

(0065) = E6

- Programa fuente en assembler

```
NAM       CARGA
ORG.        100

LDAA        $61
ADDA        $63   Suma de los bits menos significativos
STAA        $65   Guarda el resultado (bits menores)
LDAA        $60
ADCA        $62   Suma con carry de los bits más signi-
                  ficativos.
STAA        $64   Guarda el resultado (bits mayores)
SWI               Interrupción por software
```

La carga del código assembler se la puede observar en los listados mostrados en la página 212.

5.3 DEMOSTRACION DE CAPTACION DE ERRORES E IMPRESION DE MENSAJES DE ERROR.

Muchas aplicaciones requieren el reconocimiento de caracteres ASCII, Estos caracteres pueden corresponden a comandos, códigos de identificación, nombres o números. El problema del reconocimiento involucra comparar los símbolos y ver si son iguales.

- Datos de un problema de ejemplo

Supongamos que dos secuencias de caracteres ASCII comienzan en la localidad de memoria 0052 y 0062, respectivamente. La localidad 0051 contiene el número de caracteres de cada serie. El programa coloca la localidad 0050 a cero, si los caracteres son iguales y a FF si no lo son.

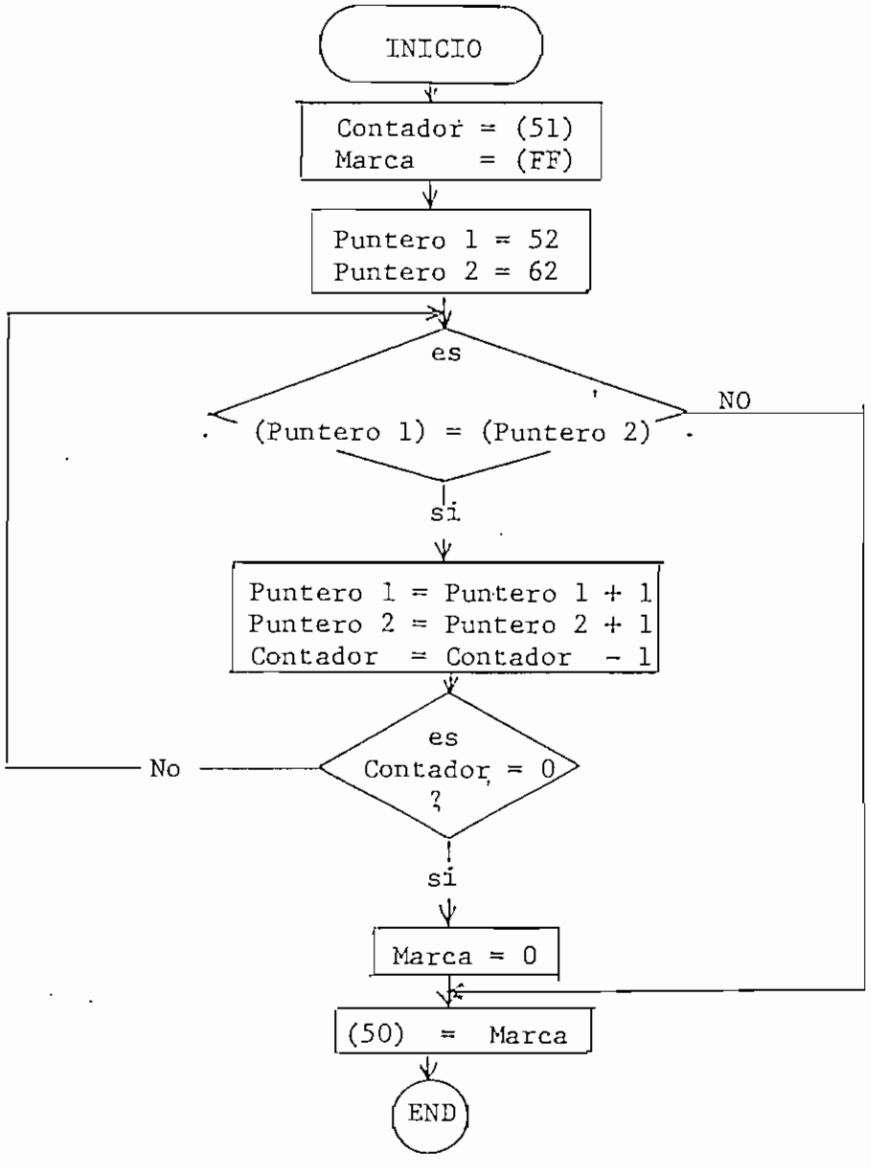
(0051) = 4
(0052) = 54 T
(0053) = 4F 0
(0054) = 4D M
(0055) = 41 A

(0062) = 54 T
(0063) = 4F O
(0064) = 44 D
(0065) = 41 A

Las dos secuencias de cuatro caracteres no es la misma, pues no corresponde el tercer carácter. La primera diferencia basta para concluir la búsqueda.

El resultado es por lo tanto: (0050) = FF.

Diagrama de flujo



El programa fuente inicial en Assembler con algunas instrucciones incorrectas, se muestra en la página 216. A continuación se puede apreciar el intento de ensamblaje del programa fuente y los mensajes de error que se producen, ésto en la página 216. Por último, el programa fuente final en Assembler se encuentra en la página 218 .

5.4 ILUSTRACION DEL USO DE SUBRUTINAS.

Cualquier programa puede ser convertido en una subrutina, para ello simplemente se etiqueta la primera instrucción como punto de entrada y se finaliza el programa con un return-from-subroutine. El programa principal debe llamar a la subrutina.

Se va a desarrollar un programa de ejemplo con una subrutina. La subrutina determina la longitud de una secuencia de caracteres ASCII. La dirección inicial de la serie está en el registro índice. El fin de la serie está marcada por el código hexadecimal OD guardado en memoria. El resultado, es decir la longitud de una secuencia, excepto el carácter de retorno, se ubica en el acumulador B.

- Problemas de ejemplo:

a. (X) = 33

(0033) = 0D

Resultado (B) = 00

b. (X) = 0033

(0033) = 4D M

(0034) = 41 A

(0035) = 59 Y

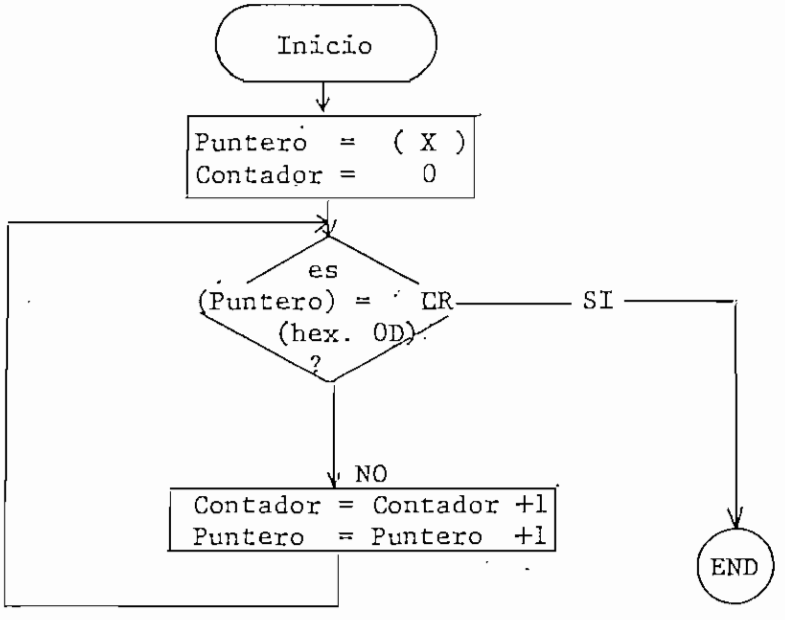
(0036) = 4F O

(0037) = 52 R

(0038) = 0D Carácter de retorno

Resultado (B) = 05

- Diagrama de flujo



- Programa principal

Guarda en el stack la localidad 0058, toma las direcciones iniciales de las localidades de memoria 0040 y 0041, llama a la subrutina que determina el número de caracteres ASCII y guarda el resultado en la dirección 0042.

El listado de este programa se lo encuentra en la página 219

Las instrucciones de la subrutina se pueden estudiar en la página 219 : El programa ensamblado y el que está en assembler, se localizan en la página 219 .

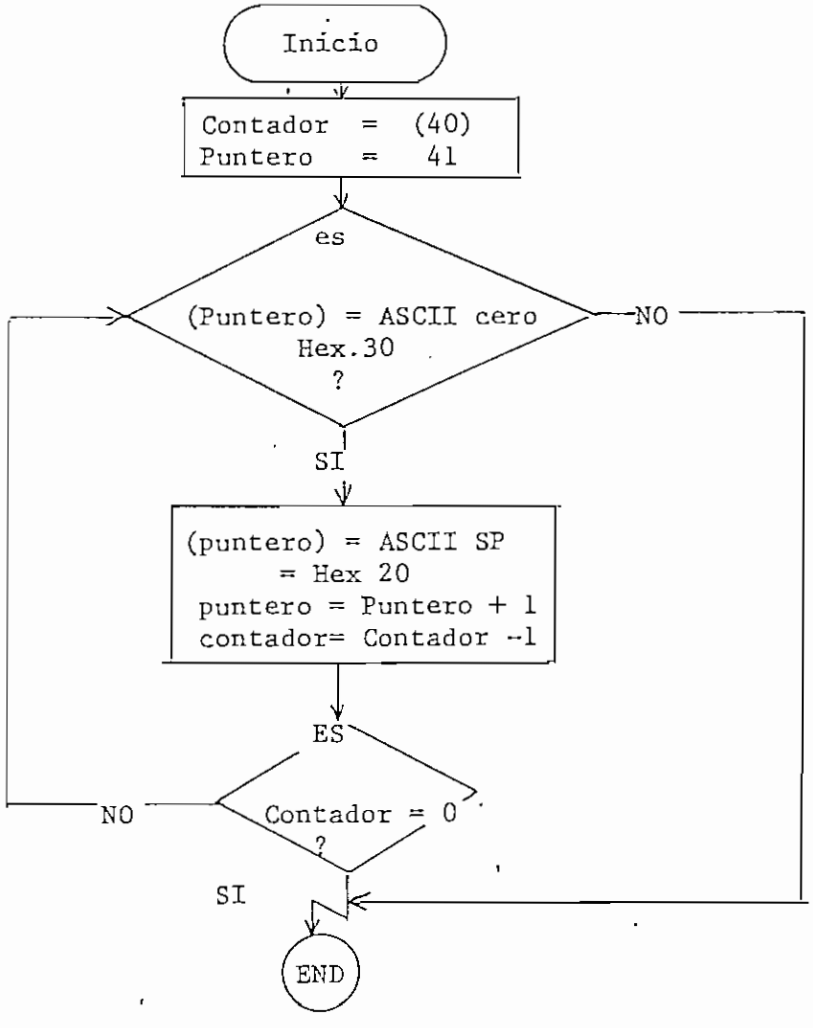
Se observa que esta subrutina tiene un solo parámetro que es una dirección. Esta se guarda en el registro índice. El resultado retorna en el acumulador B.

El programa principal tiene tres pasos fundamentales:

- Ubicar la dirección inicial de la secuencia de caracteres en el registro índice.
- Llamar la subrutina
- Guardar el resultado

La inicialización se debe realizar en el stack en una área de memoria apropiada.

- Diagrama de flujo



El listado del programa en assembler se lo puede apreciar en la página 220 al igual que el programa ya ensamblado.

En este programa el lazo tiene dos salidas, una si el procesador encuentra un dígito no cero; y, otra, si se ha examinado la secuencia entera.

El carácter ASCII cero no aparece entre apóstrofes. Se puede usar, como se menciona en el apéndice C, la instrucción FCC (form Constant Characters), para ubicar un carácter ASCII en la memoria del programa. Esto se hará así:

```
ETIQU    FCC    0
```

Debe notarse que cada carácter ASCII necesita 8 bits para su representación interna.

b. Se observará a continuación un ejemplo en el que se hace una clasificación de números binarios sin signo en orden descendente. La longitud del arreglo está en la localidad de memoria 0041 y el arreglo comienza en la dirección 0042.

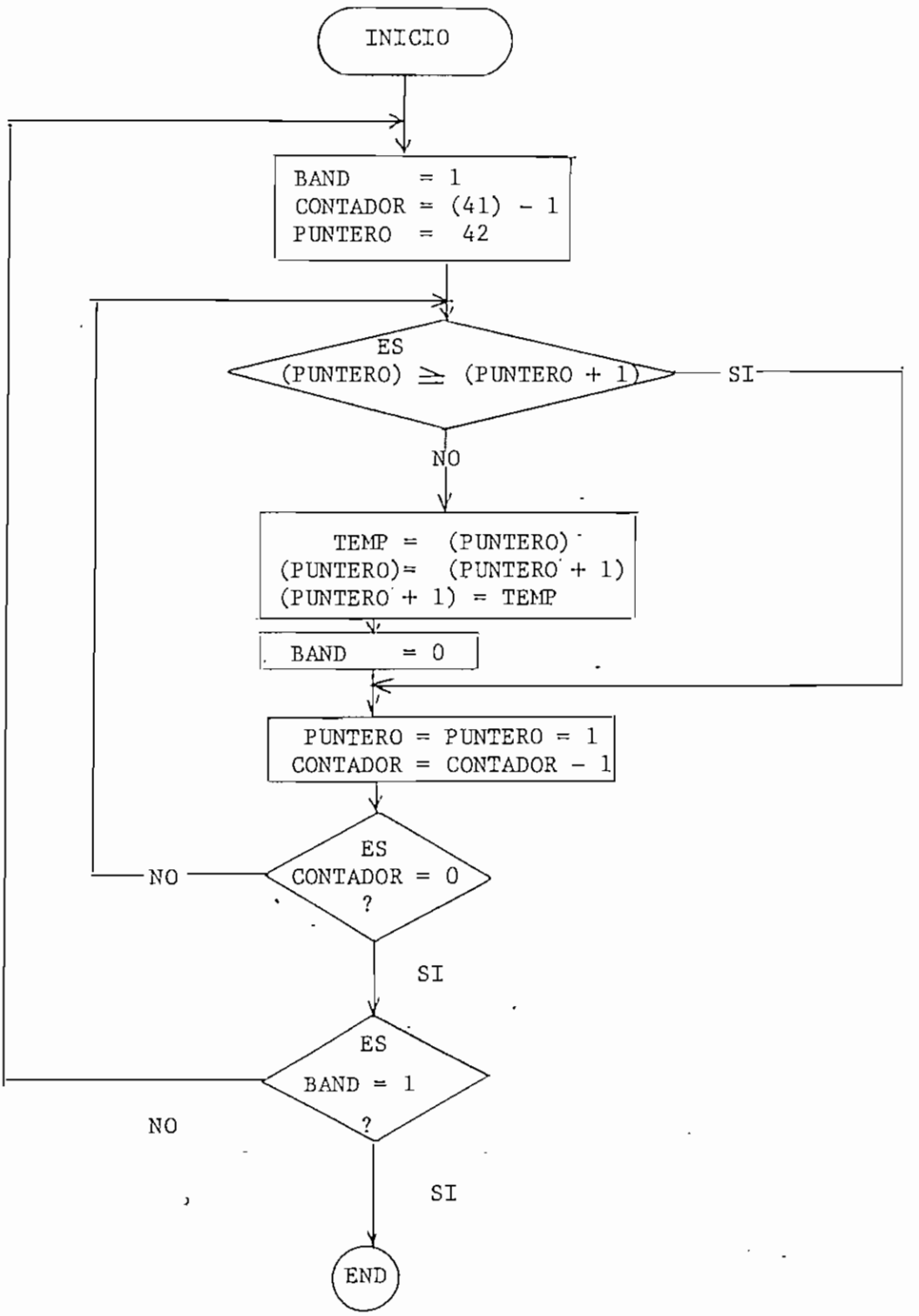
   - Datos de un problema de ejemplo:

```
(0041) = 06
(0042) = 5A
(0043) = 4F
(0044) = C5
(0045) = 10
(0046) = B1
(0047) = 2E
```

Resultado:

```
(0042) = 10
(0043) = 2E
(0044) = 4F
(0045) = 5A
(0046) = B1
(0047) = C5
```

- Diagrama de flujo.



```
M (401A 401B)

==>             ALLOCATE CARGA.S(401B) SECTORS 10
CARGA.S(401B)
CART ID 401B    DB ADDR  4910    DB CNT    00A0

==>             EDITOR

WHAT FILE  4?>      CARGA


      ( CARGA INICIAL DE PROGRAMA? (S/N) )
S


      INGRESE EL PROGRAMA
      FINALICE LA CARGA CON UNA LINEA DE '/*' EN LAS DOS PRIMERAS COLUMNAS



                          ETIQU  OPER  OPERANDO
                                 NAM   CARGA
                                 ORG   $100
                          *
                          * M.B.H.A.
                          *
                          * PROGRAMA QUE SUMA NUMEROS DE 16 BITS
                          *
                                 LDAA  $61
                                 ADDA  $63 SUMA DE BITS MENOS SIGNIF.
                                 STAA  $65 GUARDA EL RESULTADO
                                 LDAA  $60
                                 ADDA  $62 SUMA DE BITS MAS SIGNIF.
                                 STAA  $64 GUARDA EL RESULTADO
                                 SWI
                                 END
/*


      FIN DE LA GARGA


      INSERTANDO LINEAS? (S/N)
S


      INSERTE DESPUES DE LINEA XXX
                              011


                          ETIQU  OPER  OPERANDO
                          * EJEMPLO DE LINEA DE INSERCION
/*


      INSERTANDO LINEAS? (S/N)
N


      ELIMINANDO LINEAS? (S/N)
N
```

```
LISTAR UN NUMERO DE LINEAS? (S/N)
N

LISTAR EL PROGRAMA? (S/N)
S
 1:                                NAM   CARGA
 2:                                ORG   $100
 3:                          *
 4:                          * M.B.H.A.
 5:                          *
 6:                          * PROGRAMA QUE SUMA NUMEROS DE 16 BITS
 7:                          *
 8:                                LDAA  $61
 9:                                ADDA  $63 SUMA DE BITS MENOS SIGNIF.
10:                                STAA  $65 GUARDA EL RESULTADO
11:                                LDAA  $60
12:                          * EJEMPLO DE LINEA DE INSERCION
13:                                ADDA  $62 SUMA DE BITS MAS SIGNIF.
14:                                STAA  $64 GUARDA EL RESULTADO
15:                                SWI
16:                                END

LISTAR EL PROGRAMA? (S/N)
N

MODIFICAR UN REGISTRO ? (S/N)
S

INGRESE EL PUNTERO DEL REGISTRO
9999
0015

INGRESE EL NUEVO REGISTRO DEBAJO DEL ANTERIOR

                          SWI
                   AQUI   BRA   AQUI

FIN DEL TRABAJO? (S/N)
N

( CARGA INICIAL DE PROGRAMA? (S/N) )
N

INSERTANDO LINEAS? (S/N)
N

ELIMINANDO LINEAS? (S/N)
N

LISTAR UN NUMERO DE LINEAS? (S/N)
N
```

```
   LISTAR EL PROGRAMA? (S/N)
:I
    1:      00001                          NAM   CARGA
    2:      00002 0100                     ORG   $100
    3:      00003                *
    4:      00004                * M.B.H.A.
    5:      00005                *
    6:      00006                * PROGRAMA QUE SUMA NUMEROS DE 16 BITS
    7:      00007                *
    8:      00008 0100 96 61               LDAA  $61
    9:      00009 0102 9B 63               ADDA  $63 SUMA DE BITS MENOS SIGNIF.
   10:      00010 0104 97 65               STAA  $65 GUARDA EL RESULTADO
   11:      00011 0106 96 60               LDAA  $60
   12:      00012                * EJEMPLO DE LINEA DE INSERCION
   13:      00013 0108 9B 62               ADDA  $62 SUMA DE BITS MAS SIGNIF.
   14:      00014 010A 97 64               STAA  $64 GUARDA EL RESULTADO
   15:                           AQUI     BRA   AQUI
   16:      00016                          END

   LISTAR EL PROGRAMA? (S/N)


   MODIFICAR UN REGISTRO ? (S/N)


   FIN DEL TRABAJO? (S/N)
I

=>
```

```
WHAT FILE  1?>     NMONI

WHAT FILE  2?>     SFL

WHAT FILE  3?>     ASCII

WHAT FILE  4?>     CARGA

     INGRESE UN : 1 PARA SALIDA EN PANTALLA

                  5 PARA.SALIDA EN IMPRESORA

?
1
1

     ESCUELA POLITECNICA NACIONAL
     FACULTAD DE INGENIERIA ELECTRICA

     TESIS DE GRADO :  25 DE JUNIO DE 1982


     REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
     DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.


     SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
     ===============================================


     00001                        NAM   CARGA
     00002 0100                   ORG   $100
     00003                  *
     00004                  * M.B.H.A.
     00005                  *
     00006                  * PROGRAMA QUE SUMA NUMEROS DE 16 BITS
     00007                  *
     00008 0100 96 61             LDAA  $61
     00009 0102 9B 63             ADDA  $63 SUMA DE BITS MENOS SIGNIF.
     00010 0104 97 65             STAA  $65 GUARDA EL RESULTADO
     00011 0106 96 60             LDAA  $60
     00012                  * EJEMPLO DE LINEA DE INSERCION
     00013 0108 9B 62             ADDA  $62 SUMA DE BITS MAS SIGNIF.
     00014 010A 97 64             STAA  $64 GUARDA EL RESULTADO
     00015 010C 20 FE    AQUI     BRA   AQUI
     00016                        END

     TABLA DE SIMBOLOS

     AQUI     010C


     NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE ESTE PROGRAMA

     FIN DE UN ENSAMBLAJE EXITOSO

==>
```

1

ESCUELA POLITECNICA NACIONAL
FACULTAD DE INGENIERIA ELECTRICA

TESIS DE GRADO : 25 DE JUNIO DE 1982

REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.

SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
==========

```
00001                            NAM    COMCR
00002 0000                       ORG    0
00003                  *
00004                  * M.B.H.A.
00005                  *
00006                  * PROGRAMA QUE COMPARA CARACTERES.
00007                  *
00008 0064                       ORG    100
00009                  *
00010                  * PRUEBA DE PSEUDO OPERACIONES
00011                  *
00012       0009       ETIQU   EQU    %1001
00013 0064 0003       DATA1   RMB    3
00014 0067 40         DATA2   FCB    $40
00015 0068 41         DATA3   FCC    A
00016                  *
00017                  * AQUI COMIENZA EL PROGRAMA
00018                  *
00019 0000                       ORG    0
00020 0000 86 FF                 LDAA   #$FF
*E12*00021                        STAA   '$40 MARCA IGUAL A FF
*E10*00022                       LDAA   $#41 CONTADOR = LONG. DE SEC.
00023 0002 A6 00                 LDAA   X TOME UN ELEMENTO SEC. UNO
00024 0004 91 10                 CMPA   $10,  ES EL MISMO ELEM. SECUENCIA 2 ?
00025 0006 26 07                 BNE    FIN NO
00026 0008 08                    INX
00027 0009 5A                    DECB    TODOS LOS CARAC. CHEQUEADOS ?
*E05*00028 000A 26               BNE    CARAC NO, SIGA ANALIZANDO
00029 000C 7F 0040               CLR    $40 SI, MARCA IGUAL A CERO
00030 000F 3F          FIN       SWI
00031                            END
```

TABLA DE SIMBOLOS

ETIQU	0009
DATA1	0064
DATA2	0067
DATA3	0068
FIN	000F

*** 3 ERRORES DETECTADOS EN EL ENSAMBLAJE DE ESTE PROGRAMA ***

FIN DE LA SALIDA DEL CROSS-ASSEMBLER

```
ERROR

WHAT FILE  5?>     NERRO

     INGRESE EL CODIGO DE ERROR (2 DIGITOS)

DD
08
08   SIMBOLO INDEFINIDO : UN SIMBOLO USADO NO HA SIDO DEFINIDO.

     FIN DEL TRABAJO? (S/N)

     INGRESE EL CODIGO DE ERROR (2 DIGITOS)

DD
10
10   EXPRESION INVALIDA : UNA EXPRESION HA SIDO FORMADA DE UNA MANERA ILE
                          GAL. PROBLEMAS COMUNES INCLUYEN : OPERACIONES IN
                          VALIDAS ( ERRORES DE DIGITACION O SIMBOLOS INCO
                          RRECTOS ), ARGUMENTOS OMITIDOS, ARGUMENTOS DE TI
                          PO O LONGITUD NO PERMITIDA, ETC.

     FIN DEL TRABAJO? (S/N)

     INGRESE EL CODIGO DE ERROR (2 DIGITOS)

DD
12
12   CODIGO DE OPERACION INDEFINIDO : UN CODIGO DE OPERACION USADO NO SE EN
                                      CUENTRA EN EL SET DE INSTRUCCIONES.
                                      ESTE ERROR USUALMENTE SIGNIFICA QUE E
                                      CODIGO DE OPERACION ESTA MAL DELETREA
                                      DO, MAL UBICADO, U OMITIDO.

     FIN DEL TRABAJO? (S/N)
S

==>
```

ESCUELA POLITECNICA NACIONAL
FACULTAD DE INGENIERIA ELECTRICA

TESIS DE GRADO : 25 DE JUNIO DE 1982

REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.

SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
==========

```
00001                           NAM   COMCR
00002 0000                      ORG   0
00003                   *
00004                   * M.B.H.A.
00005                   *
00006                   * PROGRAMA QUE COMPARA CARACTERES.
00007                   *
00008 0064                      ORG   100
00009                   *
00010                   * PRUEBA DE PSEUDO OPERACIONES
00011                   *
00012      0009         ETIQU  EQU   %1001
00013 0064 0003         DATA1  RMB   3
00014 0067 40           DATA2  FCB   $40
00015 0068 41           DATA3  FCC   A
00016                   *
00017                   * AQUI COMIENZA EL PROGRAMA
00018                   *
00019 0000                      ORG   0
00020 0000 86 FF                LDAA  #$FF
00021 0002 97 40                STAA  $40 MARCA IGUAL A FF
00022 0004 D6 41                LDAB  $41 CONTADOR = LONG. DE SECUENCIA
00023 0006 A6 00        CARAC   LDAA  X TOME UN ELEMENTO SEC. UNO
00024 0008 A1 10                CMPA  $10,X ES EL MISMO ELEM. SECUENCIA 2 ?
00025 000A 26 07                BNE   FIN NO
00026 000C 08                   INX
00027 000D 5A                   DECB   TODOS LOS CARAC. CHEQUEADOS ?
00028 000E 26 F6                BNE   CARAC NO, SIGA ANALIZANDO
00029 0010 7F 0040              CLR   $40 SI, MARCA IGUAL A CERO
00030 0013 3F           FIN     SWI
00031                           END
```

TABLA DE SIMBOLOS

```
ETIQU   0009
DATA1   0064
DATA2   0067
DATA3   0068
CARAC   0006
FIN     0013
```

NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE ESTE PROGRAMA

FIN DE UN ENSAMBLAJE EXITOSO

1

ESCUELA POLITECNICA NACIONAL
FACULTAD DE INGENIERIA ELECTRICA

TESIS DE GRADO : 25 DE JUNIO DE 1982

REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.

SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
==========

```
00001                           NAM   EJSUB
00002 0000                      ORG   0
00003                   *
00004                   * M.B.H.A.
00005                   *
00006                   * PROGRAMA QUE DETERMINA LA LONGITUD DE UNA
00007                   * SECUENCIA DE CARACTERES. SIRVE COMO EJEM-
00008                   * PLO DEL USO DE SUBRUTINAS.
00009                   *
00010                   * PROGRAMA PRINCIPAL :
00011                   *
00012 0000 9F 3E                STS   $3E GUARDE EL PUNTERO DEL STACK
00013 0002 8E 0058              LDS   #$58 INICIALIZAR STACK A 0058
00014 0005 DE 40                LDX   $40 TOMA DIREC. INICIAL DE SEC.
00015 0007 BD 0020              JSR   LONGI DETERMINA LONG. DE SEC.
00016 000A D7 42                STAB  $42 GUARDA LONG. DE LA SEC.
00017 000C 9E 3E                LDS   $3E RECUPERA EL PUNT. DEL STACK
00018 000E 3F                   SWI
00019                   *
00020                   * SUBRUTINA LONGI :
00021                   *
00022 0020                      ORG   $20
00023 0020 5F           LONGI   CLRB   LONGITUD DE LA SECUENCIA = 0
00024 0021 86 0D                LDAA  #$0D CARGAR CODIGO FIN CARAC.
00025 0023 A1 00        COMP    CMPA  X ES FIN DE CARACTERES ?
00026 0025 27 04                BEQ   FIN SI, FIN DE CARACTERES
00027 0027 5C                   INCB   NO, SUME UNO A LA LONG. DE SEC.
00028 0028 08                   INX
00029 0029 20 F8                BRA   COMP
00030 002B 39           FIN     RTS
00031                           END
```

TABLA DE SIMBOLOS

```
LONGI    0020
COMP     0023
FIN      002B
```

NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE ESTE PROGRAMA

FIN DE UN ENSAMBLAJE EXITOSO

==>

ESCUELA POLITECNICA NACIONAL
FACULTAD DE INGENIERIA ELECTRICA

TESIS DE GRADO : 25 DE JUNIO DE 1982

REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.

SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
==========

```
00001                           NAM   COMBL
00002 0000                      ORG   0
00003                   *
00004                   * M.B.H.A.
00005                   *
00006                   * PROGRAMA QUE REEMPLAZA CEROS CON BLANCOS
00007                   * EN UNA SECUENCIA DE CARACTERES ASCII
00008                   * (DIGITOS DECIMALES)
00009                   *
00010 0000 86 30                LDAA  #$30 CARGA COD. CARAC. CERO
00011 0002 C6 20                LDAB  #$20 CARGA COD. CARAC. BLANCO
00012 0004 CE 0041              LDX   #$41 PUNTERO = INIC. DEL ARREGLO
00013 0007 A1 00        CONT    CMPA  X ES ELEMENTO ASCII CERO ?
00014 0009 26 08                BNE   FIN NO, FINALICE EL TRABAJO
00015 000B E7 00                STAB  X SI, REEMPLAZA CERO CON BLANCOS
00016 000D 08                   INX
00017 000E 7A 0040              DEC   $40
00018 0011 26 F4                BNE   CONT
00019 0013 3F           FIN     SWI
00020                           END
```

TABLA DE SIMBOLOS

```
CONT    0007
FIN     0013
```

NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE ESTE PROGRAMA

FIN DE UN ENSAMBLAJE EXITOSO

==>

ESCUELA POLITECNICA NACIONAL
FACULTAD DE INGENIERIA ELECTRICA

TESIS DE GRADO : 25 DE JUNIO DE 1982

REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ
DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.

SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA M6800
==========

```
00001                           NAM   ORDEN
00002 0000                      ORG   0
00003                   *
00004                   * M.B.H.A.
00005                   *
00006                   * ORDENAR UN ARREGLO DE NUMEROS BINARIOS SIN
00007                   * SIGNO EN ORDEN DESCENDENTE.
00008                   *
00009 0000 86 01        ORDE    LDAA  #1 BANDERA IGUAL A 1
00010 0002 97 40                STAA  $40
00011 0004 96 41                LDAA  $41 AJUSTA LONG. DEL ARREGLO A # DE PARES
00012 0006 4A                   DECA
00013 0007 CE 0042              LDX   #$42 APUNTA AL PRINCIPIO DEL ARREGLO
00014 000A E6 00        PASO    LDAB  X
00015 000C E1 01                CMPB  1,X ESTA EL PAR EN ORDEN?
00016 000E 24 0B                BCC   CONT SI, TOME EL SIGUIENTE PAR
00017 0010 7F 0040              CLR   $40 NO, BORRE BANDERA
00018 0013 36                   PSHA   GUARDE CONTADOR DEL ARREGLO
00019 0014 A6 01                LDAA  1,X INTERCAMBIE ELEMENTOS FUERA DE ORDEN
00020 0016 E7 01                STAB  1,X
00021 0018 A7 00                STAA  X
00022 001A 32                   PULA   RECUPERE EL CONTADOR DEL ARREGLO
00023 001B 08           CONT    INX
00024 001C 4A                   DECA   DECREMENTE EL CONTADOR EN 1
00025 001D 26 EB                BNE   PASO
00026 001F 7D 0040              TST   $40 TODOS LOS ELEMENTOS EN ORDEN ?
00027 0022 27 DC                BEQ   ORDE NO, PROCESE EL ARREGLO OTRA VEZ
00028 0024 3F                   SWI
00029                           END
```

TABLA DE SIMBOLOS

```
ORDE    0000
PASO    000A
CONT    001B
```

NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE ESTE PROGRAMA

FIN DE UN ENSAMBLAJE EXITOSO

CAPITULO VI

CONCLUSIONES Y PROYECCIONES DEL CROSS-ASSEMBLER DESARROLLADO.

El Cross-Assembler desarrollado en el presente trabajo, es una herramienta de muchísima utilidad en el desarrollo de sistemas de software para el microprocesador 6800 de la Motorola.

Desarrollar programas para microprocesadores sin contar con un ensamblador, permite sólo una alternativa: realizar la programación en assembler y luego traducir instrucción por instrucción a lenguaje de máquina (ensamblaje manual) ó, programar en objeto con todos los incovenientes y dificultades que ésto implica.

En el enfrentamiento de esta realidad, es cuando este compilador puede ser de gran ayuda, al permitir que se programe directamente en el assembler de la M6800 y facilitar las depuraciones y pruebas de programas en el minicomputador GA-DM-250 , posibilitando la obtención de un código objeto óptimo que simplemente deba ser cargado en la memoria del procesa-

dor para ser ejecutado.

Se ha usado el minicomputador DM-250 de la General Automation con que cuenta el Centro de Cómputo del IETEL, por cuanto presta la posibilidad de trabajar en forma interactiva conversacional que da flexibilidad en la utilización del Cross-Compilador y posibilita la tarea de los sistemas de edición del mismo.

Por otro lado, acogiéndose al convenio de colaboración existente entre el IETEL y la EPN, se ha visto que hay la posibilidad de usar el ensamblador en el IETEL, por parte de profesores y estudiantes de la EPN que necesiten desarrollar sistemas para la M6800, ésto sería posible previa una etapa de coordinación con las autoridades del Centro de Cómputo del IETEL.

La configuración del mencionado Centro de Cómputo a breves rasgos, es la siguiente:

- Una unidad central de procesamiento de 64K palabras (1 palabra = 2 bytes = 16 bits).
- 16 líneas para trabajar con terminales con procesos iterativos.
- 3 unidades de discos removibles de 80MB cada una.
- Unidades de: impresión, cintas magnéticas, lectoras

de tarjetas y de cinta perforada de papel.

El cross-compilador tiene salidas en pantalla de terminal,en terminal impresora (papel) o en una impresora en línea de alta velocidad (papel).

La entrada de los programas a ser ensamblados se la realiza por terminal, que es el medio que hace más versátil el trabajo del compilador y sus programas tutores de edición. Sinembargo, después de ligeras modificaciones en los programas, también podría hacerse el ingreso de datos por algún otro tipo de periférico como: lectora de tarjetas, cinta magnética, etc.

Como una proyección del compilador, se podría pensar en hacer que muchas de las subrutinas que se usan repetidamente en su ejecución se pueden programar en el assambler de la máquina DM-250 y no en FORTRAN. Esto se lo haría para optimizar tiempo y conseguir mayor eficiencia. Sinembargo, dados los alcances que tiene el Cross-ensamblador aquí desarrollado, resulta más conveniente la programación en un lenguaje de alto nivel como el FORTRAN, el cual ha sido usado en esta tesis, porque de esa manera los programas en el futuro pueden ser adaptados más facilmente a otro ordenador.

Además, los programas están organizados en forma modular, distribuyéndose tareas específicas a las distintas subrutinas, lo que redunda en que se pueden realizar modificaciones con gran facilidad. Además de que permite la adaptación a otro computador, ésta estructura hace más sencillo el aumentar las posibilidades de detección de errores y las capacidades del compilador.

Este trabajo también puede ser útil para quien desee consultar sobre teoría de diseño de compiladores, pues da una visión general y bastante completa de cómo está constituído un compilador, sus fases, qué realiza cada una de ellas, cómo pueden interrelacionarse y funcionar, de modo que un traductor resulte el más apropiado para cierto tipo de aplicación.

Todas las pruebas realizadas con el cross-assembler y todas sus salidas, han sido altamente satisfactorias, realizándose en forma precisa el ensamblaje de programas. Se puede afirmar, por tanto, que se han logrado todos los objetivos propuestos.

APENDICE A

MANUAL DE UTILIZACION DEL SISTEMA CROSS-ASSEMBLER PARA LA M6800.

INSTALACION : GA-DM-250

A.1 PROCEDIMIENTOS:

Creación de archivos en disco (comando conversacional):
ALLOCATE NOMBRE DEL ARCHIVO. S (volumen-de-disco-
Sectors # Sectores.
Montar discos (comando conversacional), con un máximo de 5 discos: M (volumen-disco 1, Volumen disco 2)

A.2 LOCALIZACION DE PROGRAMAS Y ARCHIVOS

a. Programas en el disco 4000:
   - Programas en objeto e imagen de memoria: 401A
   - Programas fuente : : 401C

b. Archivos: en el disco 4000:
   - NMONI: archivo de códigos nmotécnicos y sus hexadecimales: 401B
   - SFL: archivo de etiquetas: 401B
   - ASCII: archivo de códigos ASCII: 401B

ojo Existe un error en la instrucción 'BRANCH

- Archivos de programas a ser ensamblados: 401B

Los programas y archivos pueden ser relocalizados en los discos,de acuerdo a la conveniencia del IETEL, sin que ésto cause problemas en la utilización de la tesis.

A.3 DESCRIPCION DE LA UTILIZACION DE LOS PROGRAMAS PRINCIPALES.

A.3.1. EDITOR:

Editor de programas a ser ensamblados.

Forma de utilización:

a. Montar los discos a utilizarse 401A y 401B.
b. Para el uso del programa debe haber un área de disco asignada para el programa a ensamblarse. Por lo tanto, primero se debe definir este archivo con el comando del sistema conversacional ALLOCATE: NOMBRE DEL-PROGRAMA.S (410B) SECTORS #-DE-SECTORES.

El número de sectores se calcula así:

$$\#\ Sectores = \frac{\#\ líneas\ del\ programa}{2} + 1$$

El número de sectores asignados al programa se redondea sumando un sector para garantizar que haya el espacio necesario en memoria"

c. EDITOR (pulsar NEW/LINE en el teclado del terminal)

d. Seguir las instrucciones dadas por el Editor LPR.

(Ver pág.)

A.3.2. A6800: Cross-ensamblador para el microprocesador M6800.

Forma de utilización:

a. Montar los discos de programas y archivos necesarios: 401A y 401B.

b. A6800 (pulsar NEW/LINE) en el teclado del terminal.

c. Asignar el archivo NMONI al File 1.

d. Asignar el archivo SFL al File 2.

c. Asignar el archivo ASCII al File 3.

f. Asignar el archivo del programa a ensamblarse al File 4.

g. Se ingresa un: 1 para salida en pantalla.
5 para salida en impresora

h. Se obtiene la salida del cross-ensamblador en pantalla ó en papel.

(Ver pág.).

A.3.3 TANEM: Programa para cargar los códigos nmotécnicos y los correspondientes códigos hexadecimales.

Forma de utilización:

a. Montar los discos de programas y archivos necesarios: 401A y 401B
b. TANEM (pulsar NEW/LINE en el teclado del terminal)
c. Asignar el archivo NMONI al File 1.
d. Seguir las instrucciones del programa.

(Ver pág.).

A.3.4 LASCI: Programa para cargar los códigos ASCII y el valor entero que les corresponde.

Forma de utilización:

a. Montar los discos a utilizarse: 401A y 401 B
b. Para el uso del programa debe haberse asignado un archivo ASCII que permita guardar todos los códigos ASCII. Si aún no se ha definido el archivo, se lo debe hacer con el comando ALLOCATE.

   ALLOCATE ASCII.S (401B) SECTORS 5.

c. LASCI (pulsar NEW/LINE en el teclado del terminal)
d. Asignar el archivo ASCII al File 3.
e. Seguir las instrucciones del programa.

(Ver pág.).

A.3.5 LERRO: Programa para cargar el archivo en disco que corresponde a los mensajes de error.

Forma de utilización:

a. Montar los discos a utilizarse: 401A y 401B.
b. Si no se ha asignado todavía un área en disco para el archivo, utilizar el comando ALLOCATE. Para aproximadamente unas 100 líneas con 160 palabras (80 posiciones) cada una necesitamos 50 sectores.
   ALLOCATE NERRO.S (401B) SECTORS 50.
c. LERRO (pulsar NEW/LINE en el teclado del terminal)
d. Asignar el archivo NERRO al File 5.
e. Seguir las instrucciones del programa.

A.3.6 ERROR: Programa al que se le entrega el código del error y devuelve el mensaje de error que le corresponde.

Forma de utilización:

a. Montar los discos que se van a usar: 401A y 401B.
b. ERROR(pulsar NWE/LINE en el teclado del terminal)
c. Asignar el archivo NERRO al File 5.
d. Seguir las instrucciones del programa

(Ver pág.)

```
1:*EXTENDED PRECISION
2:*TWO WORD INTEGERS
3:C
4:C      PROGRAMA : A6800
5:C
6:C      PROGRAMA PRINCIPAL PARA EL ENSAMBLADOR DE PROGRAMAS ESCRITOS
7:C      EN LENGUAJE ASSEMBLER DE LA MOTOROLA M6800.
8:C
9:C
10:C     DEFINIR LOS ARCHIVOS A UTILIZARSE, ADEMAS DEFINIR ARREGLOS E
11:C     INICIALIZAR VARIABLES.
12:C
13:      DEFINE FILE 1(400,32,U,J1),2(500,12,U,J2)
14:      DEFINE FILE 3(250,6,U,J3),4(1000,160,U,J4)
15:      INTEGER*4 ORG(3),LCMP(3)
16:      DIMENSION L80(80),LD5(5),IH4H(4)
17:      DIMENSION LTS(17),L18(18),LCOD(16),LC6(6)
18:      DIMENSION LNET(5),LBL5(5),LB18(18)
19:      DATA LTS(1)-(10)/'T','A','B','L','A',' ','D','E',' ','S'/
20:      DATA LTS(11)-(17)/'I','M','B','O','L','O','S'/
21:      DATA NAST,IBLK/'*',' '/,ORG/'O','R','G'/,IBLX/'X'/
22:C
23:C     SECUENCIALIZAR LAS INSTRUCCIONES DEL PROGRAMA,
24:C     Y BORRAR LAS AREAS DEL PROGRAMA OBJETO.
25:C
26:      CALL CGPR
27:C
28:C     INICIALIZAR EL ARCHIVO DE ETIQUETAS, Y LEER EL NUMERO DE INSTR.
29:C     DEL PROGRAMA MAS UNO
30:C
31:      NREC=1
32:      WRITE(2'1)NREC
33:      READ(4'1)IP
34:C
35:C     CONSIDERAR LOS "DIRECTIVES" : NAM Y ORG EN ESE ORDEN.
36:C
37:      READ(4'2)L80
38:      CALL VDF(L80,NTY)
39:      IF(NTY.EQ.4)GO TO 4080
40:      IT=2
41:      CALL PERR(L80,5,IT)
42:4080  DO 300 NSH=1,2
43:      READ(4'3)L80
44:      CALL VDF(L80,NTY)
45:      IF(NTY.EQ.1)GO TO 4081
46:      IT=3
47:      CALL PERR(L80,6,IT)
48:      NQ4=0
49:      GO TO 4085
50:4081  CALL SRORG(L80,NQ4,NER,3)
51:4085  IPM1=IP-1
52:      IF(NQ4.LE.65535)GO TO 4086
53:      CALL PERR(L80,13,3)
54:4086  CONTINUE
55:C
56:C     ANALIZAR EL RESTO DEL PROGRAMA ( A PARTIR DE LA LINEA NUMERO
57:C     TRES HASTA LA LINEA DEL PROGRAMA QUE ESTA ANTES DEL "END" ).
58:C
59:      DO 300 I=4,IPM1
60:      IT=I
61:      IF(NQ4.LE.65535)GO TO 4087
62:      CALL PERR(L80,13,IT)
63:4087  CONTINUE
```

```
C      CONSIDERAR EL BLOQUE DE "DIRECTIVES" :
C      ORG, EQU, RMB, FCB, Y FCC.
C
       READ(4'I)L80
C
C      ETIQUETA NO DEBE COMENZAR CON CARACTER 'X'.
C
       IF(L80(25).NE.IBLX)GO TO 4088
       CALL PERR(L80,2,IT)
4088   CONTINUE
C
C      UN ASTERISCO ( '*' ) EN LA COLUMNA 25 REPRESENTA UNA LINEA
C      DE COMENTARIO EN EL PROGRAMA DE ASSEMBLER.
C
       IF(L80(25).EQ.NAST)GO TO 300
       CALL VDF(L80,NTY)
       IF(NTY.EQ.0)GO TO 400
       IF(NTY.EQ.2)GO TO 410
       IF(NTY.EQ.3)GO TO 420
       IF(NTY.EQ.5)GO TO 999
       IF(NTY.EQ.6)GO TO 4040
       IF(NTY.EQ.7)GO TO 4030
C
C      "PAUSE" SI NO ES NINGUNO DE LOS "DIRECTIVES" PREVISTOS.
C
       IF(NTY.NE.1)PAUSE
C
C      NTY=1, "DIRECTIVE" ORG.
C
       CALL SRORG(L80,NQ4,NER,IT)
       GO TO 300
410    CALL SREQU(L80,NQ4,NER,IT,NSH)
       GO TO 300
420    CALL SRRMB(L80,NQ4,NER,IT,NSH)
       GO TO 300
4030   CALL SRFCB(L80,NQ4,NER,IT,NSH)
       GO TO 300
4040   CALL SRFCC(L80,NQ4,NER,IT,NSH)
       GO TO 300
C
C      ENSAMBLAR LAS INSTRUCCIONES DEL PROGRAMA.
C
400    CONTINUE
       IF(L80(35).NE.IBLK)GO TO 482
       CALL MOVER(L80,32,34,LCMP,1)
       CALL CMP(LCMP,ORG,NMORG)
       IF(NMORG.EQ.0)GO TO 482
       CALL PERR(L80,1,IT)
       GO TO 300
482    CONTINUE
C
C      VER SI EL CODIGO DE OPERACION ES VALIDO.
C
       CALL VNMOP(L80,LCOD,NER)
       IF(NER.NE.0)GO TO 4001
       CALL PERR(L80,12,IT)
       GO TO 300
C
C      AVERIGUAR TIPO DE DIRECCIONAMIENTO.
C
4001   CALL MOVER(L80,38,55,L18,1)
       CALL NTYPE(L18,NC,NDEQ,NCM,NPP)
       CALL LABEL(L18,NE,NDEC,LNET)
C
C      LLENAR LA TABLA DE SIMBOLOS
```

```
130:C
131:        CALL MOVER(L80,25,42,LB18,1)
132:        CALL LABEL(LB18,NBE18,NBDEC,LBL5)
133:        IF(NBE18.EQ.0)GO TO 4004
134:        IF(NBE18.EQ.1)GO TO 4003
135:        CALL PERR(L80,2,IT)
136:        GO TO 4004
137:4003    IF(NSH.EQ.2)GO TO 4004
138:        READ(2'1)NRLB
139:        NRLB=NRLB+1
140:        IF(NRLB.LE.520)GO TO 4005
141:        CALL PERR(L80,13,IT)
142:        GO TO 4004
143:4005    WRITE(2'NRLB)LBL5,NQ4
144:        WRITE(2'1)NRLB
145:4004    CONTINUE
146:C
147:        CALL INMED(L18,NX)
148:        IF(NX.EQ.1)GO TO 4002
149:        CALL RELAT(L80,LCOD,NE,NX)
150:        IF(NX.EQ.4)GO TO 4002
151:        IF(NX.EQ.6)GO TO 4002
152:        CALL INDEX(L18,NC,NDEQ,NCM,NPP,NX)
153:        IF(NX.EQ.3)GO TO 4002
154:        CALL DIREC(NDEQ,NE,NC,NX)
155:        IF(NX.EQ.2.OR.NX.EQ.4)GO TO 4002
156:        CALL IMPLI(L18,NC,LCOD,NX)
157:        IF(NX.EQ.5)GO TO 4002
158:        CALL PERR(L80,10,IT)
159:        GO TO 300
160:4002    CONTINUE
161:        CALL LCAR(LC6,LCOD,NX)
162:        L80(17)=LC6(1)
163:        L80(18)=LC6(2)
164:        WRITE(4'IT)L80
165:C
166:C       COMPLETAR CODIGOS DE OPERANDO (PRIMERA CORRIDA).
167:C
168:        GO TO (4021,4022,4023,4022,4025,4026),NX
169:4021    CONTINUE
170:        CALL COD1(L80,NQ4,IT)
171:        GO TO 300
172:4022    CONTINUE
173:        CALL COD24(L80,NQ4,LCOD,NSH,IT)
174:        GO TO 300
175:4023    CONTINUE
176:        CALL COD3(L80,NQ4,NSH,IT)
177:        GO TO 300
178:4025    CONTINUE
179:        CALL COD5(L80,NQ4,NSH,IT)
180:        GO TO 300
181:4026    CONTINUE
182:        CALL COD6(L80,NQ4,NSH,IT)
183:300     CONTINUE
184:        READ(4'IP)L80
185:        CALL VDF(L80,NTY)
186:        IF(NTY.EQ.5)GO TO 999
187:        CALL PERR(L80,1,IP)
188:C
189:C       TITULARES DE LA SALIDA DEL CROSS-ASSEMBLER
190:C
191:999     WRITE(1,257)
192:257     FORMAT(/,5X,'INGRESE UN : 1 PARA SALIDA EN PANTALLA',//,
193:       & 18X,'5 PARA SALIDA EN IMPRESORA',//,'9')
194:        READ(6,10,END=88)NWP
195:10      FORMAT(I1)
```

```
197:555     FORMAT(1H1,/,5X,'ESCUELA POLITECNICA NACIONAL',/,
198:       6 5X,'FACULTAD DE INGENIERIA ELECTRICA',//,
199:       6 5X,'TESIS DE GRADO :  25 DE JUNIO DE 1982',///,
200:       6 5X,'REALIZADO POR : MIRIAM HERNANDEZ ALVAREZ',/,
201:       6 5X,'DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.',/)
202:        WRITE(NWP,219)
203:219     FORMAT(//,5X,'SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA',
204:       6 ' M6800',/,5X,47('='),//)
205:C       CREACION DE LA TABLA DE SIMBOLOS EN EL ARCHIVO FUENTE
206:        DO 310 I=2,IP
207:        READ(4'I)L80
208:        WRITE(NWP,260)L80
209:260     FORMAT(80A1)
210:310     CONTINUE
211:        WRITE(NWP,261)
212:261     FORMAT(/,5X,'TABLA DE SIMBOLOS',/)
213:        READ(2'1)NREC
214:        DO 320 I=2,NREC
215:        READ(2'I)LD5,IHD5
216:        CALL DTH(IHD5,IH4H)
217:        CALL NTAAC(IH4H,1,4)
218:        WRITE(NWP,262)LD5,IH4H
219:262     FORMAT(5X,5A1,3X,4A1)
220:320     CONTINUE
221:        NERCT=0
222:        DO 330 I=2,IP
223:        READ(4'I)L80
224:        IF(L80(1).EQ.NAST)NERCT=NERCT+1
225:330     CONTINUE
226:        IF(NERCT.EQ.0)GO TO 4082
227:        WRITE(NWP,22)NERCT
228:22      FORMAT(/,5X,'*** ',I4,' ERRORES DETECTADOS EN EL ENSAMBLAJE',
229:       6 ' DE ESTE PROGRAMA    ***')
230:        GO TO 4083
231:4082    WRITE(NWP,233)
232:233     FORMAT(//,5X,'NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE',
233:       6 ' ESTE PROGRAMA',//,5X,'FIN DE UN ENSAMBLAJE EXITOSO',/)
234:        GO TO 88
235:4083    WRITE(NWP,221)
236:221     FORMAT(/,5X,'FIN DE LA SALIDA DEL CROSS-ASSEMBLER',/)
237:88      CALL EXIT
238:        END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C
 5:C      PROGRAMA : EDITOR
 6:C
 7:C      PROGRAMA PARA DAR LAS FACILIDADES DE EDICION DE UN PROGRAMA EN
 8:C      ASSEMBLER DE LA MOTOROLA M6800. ESTE PROGRAMA PUEDE LUEGO SER
 9:C      ENSAMBLADO USANDO EL ENSAMBLADOR A6800 QUE FORMA PARTE DE ESTA
10:C      TESIS.
11:C
12:C
13:       DEFINE FILE 4(1000,160,U,J4)
14:       DIMENSION L(80),LA(80)
15:       DATA NS/'S'/
16:411    WRITE(1,200)
17:200    FORMAT(//,5X,'( CARGA INICIAL DE PROGRAMA? (S/N) )')
18:       READ(6,80)IND
19:80     FORMAT(1A1)
20:       IF(IND.NE.NS)GO TO 400
21:       IP=1
22:       WRITE(4'1)IP
23:       READ(4'1)IP
24:       IP=IP+1
25:       WRITE(1,201)
26:201    FORMAT(//,5X,'INGRESE EL PROGRAMA',/,5X,
27:      6 'FINALICE LA CARGA CON UNA LINEA DE ''/*''',
28:      6 ' EN LAS DOS PRIMERAS COLUMNAS',//)
29:       WRITE(1,220)
30:220    FORMAT(//,24X,'ETIQU  OPER  OPERANDO')
31:401    READ(6,100,END=91)L
32:100    FORMAT(80A1)
33:       WRITE(4'IP)L
34:       IP=IP+1
35:       GO TO 401
36:91     IP=IP-1
37:       WRITE(4'1)IP
38:       WRITE(1,202)
39:202    FORMAT(//,5X,'FIN DE LA GARGA',/)
40:400    WRITE(1,203)
41:203    FORMAT(/,5X,'INSERTANDO LINEAS? (S/N)')
42:       READ(6,80)IND
43:       IF(IND.NE.NS)GO TO 402
44:403    WRITE(1,204)
45:204    FORMAT(//,5X,'INSERTE DESPUES DE LINEA XXX')
46:       READ(6,105)NL
47:105    FORMAT(30X,I3)
48:       NL=NL+1
49:       READ(4'1)IP
50:       IF(NL.LT.1.OR.NL.GT.IP)GO TO 403
51:       WRITE(1,220)
52:420    READ(6,100,END=400)LA
53:       I1=IP-NL
54:       IP=IP+1
55:       NL=NL+1
56:       K2=IP
57:       NCT=0
58:       DO 310 K=NL,IP
59:       NCT=NCT+1
60:       K1=IP-NCT
61:       READ(4'K1)L
62:       WRITE(4'K2)L
63:       K2=K1
```

```
64:310    CONTINUE
65:       WRITE(4'NL)LA
66:       WRITE(4'1)IP
67:       GO TO 420
68:402    WRITE(1,205)
69:205    FORMAT(//,5X,'ELIMINANDO LINEAS? (S/N)')
70:       READ(6,80)IND
71:       IF(IND.NE.NS)GO TO 405
72:       READ(4'1)IP
73:406    WRITE(1,206)
74:206    FORMAT(//,5X,'ELIMINE XXX LINEAS DESDE LINEA XXX')
75:       READ(6,101)NDL,NL
76:101    FORMAT(13X,I3,20X,I3)
77:       NL=NL+1
78:       IF(NDL.LE.0)NDL=1
79:       IF(NL.LT.2.OR.NL.GT.IP)GO TO 406
80:       NPLL=NDL+NL-1
81:       IF(NPLL.GT.IP)NPLL=IP
82:       I1=NPLL+1
83:       IF(I1.GT.IP)GO TO 410
84:       I2=NL-1
85:       DO 300 I=I1,IP
86:       I2=I2+1
87:       READ(4'I)L
88:       WRITE(4'I2)L
89:300    CONTINUE
90:410    IP=IP-NDL
91:       WRITE(4'1)IP
92:       GO TO 402
93:405    READ(4'1)IP
94:409    WRITE(1,207)
95:207    FORMAT(//,5X,'LISTAR UN NUMERO DE LINEAS? (S/N)')
96:       READ(6,80)IND
97:       IF(IND.NE.NS)GO TO 407
98:408    WRITE(1,208)
99:208    FORMAT(//,5X,'LISTE XXX LINEAS DESDE LINEA XXX')
100:      READ(6,102)NDL,NL
101:102   FORMAT(11X,I3,20X,I3)
102:      NL=NL+1
103:      IF(NL.LT.2.OR.NL.GT.IP)GO TO 408
104:      IF(NDL.LT.1)NDL=1
105:      NPLL=NDL+NL-1
106:      IF(NPLL.GT.IP)NPLL=IP
107:      DO 301 I=NL,NPLL
108:      JX=I-1
109:      READ(4'I)L
110:      WRITE(1,209)JX,L
111:209   FORMAT(I5,':',80A1)
112:301   CONTINUE
113:      GO TO 409
114:407   WRITE(1,210)
115:210   FORMAT(//,5X,'LISTAR EL PROGRAMA? (S/N)')
116:      READ(6,80)IND
117:      IF(IND.NE.NS)GO TO 416
118:      READ(4'1)IP
119:      IF(IP.EQ.1)GO TO 407
120:      DO 302 I=2,IP
121:      READ(4'I)L
122:      J=I-1
123:      WRITE(1,209)J,L
124:302   CONTINUE
125:      GO TO 407
126:416   WRITE(1,227)
127:227   FORMAT(//,5X,'MODIFICAR UN REGISTRO ? (S/N)')
128:      READ(6,80)IND
129:      IF(IND.NE.NS)GO TO 415
```

```
130:417     WRITE(1,221)
131:221     FORMAT(//,5X,'INGRESE EL PUNTERO DEL REGISTRO',/,'9999')
132:        READ(6,170,END=415)JJ
133:170     FORMAT(I4)
134:        JJ=JJ+1
135:        IF(JJ.LE.1)GO TO 417
136:        WRITE(1,223)
137:223     FORMAT(//,5X,'INGRESE EL NUEVO REGISTRO DEBAJO DEL ANTERIOR',/)
138:        READ(4'JJ)L
139:        WRITE(1,222)L
140:222     FORMAT(80A1)
141:        READ(6,222)L
142:        WRITE(4'JJ)L
143:415     WRITE(1,211)
144:211     FORMAT(//,5X,'FIN DEL TRABAJO? (S/N)')
145:        READ(6,80)IND
146:        IF(IND.NE.NS)GO TO 411
147:        CALL EXIT
148:        END
> LAST LINE
>
```

```
*EXTENDED PRECISION
*TWO WORD INTEGERS
C
C       PROGRAMA : LERRO
C
C       PROGRAMA DE CARGA Y LECTURA DEL ARCHIVO 5 : NERRO
C       NERRO ES EL ARCHIVO CON LA DESCRIPCION DE LOS ERRORES.
C
        DEFINE FILE 5(100,160,U,J5)
        DIMENSION NERRO(80)
        DATA ISN/'S'/
        DATA IBLK/' '/
        WRITE(1,100)
100     FORMAT(//,5X,'INICIALIZACION DEL ARCHIVO ? (S/N)')
        READ(6,200)NS
200     FORMAT(A1)
        IF(NS.NE.ISN)GO TO 401
        DO 301 I1=1,80
301     NERRO(I1)=IBLK
        JJ=0
        WRITE(5'1)JJ
        DO 303 I2=2,100
303     WRITE(5'I2)NERRO
401     CONTINUE
        WRITE(1,101)
101     FORMAT(//,5X,'LECTURA DEL ARCHIVO NERRO ? (S/N)')
        READ(6,200)NS
        IF(NS.NE.ISN)GO TO 400
        READ(5'1)NREG
        WRITE(1,109)NREG
109     FORMAT(I3)
        IF(NREG.EQ.0)GO TO 400
        WRITE(1,102)
102     FORMAT(//,'ER    ','SIGNIFICADO DEL CODIGO DE ERROR',
       6 44('.'),/)
        NREG=NREG+1
        DO 300 J=2,NREG
        READ(5'J)NERRO
        WRITE(1,120)NERRO
300     CONTINUE
        NREG=NREG-1
        WRITE(5'1)NREG
400     CONTINUE
        WRITE(1,104)
104     FORMAT(//,5X,'CARGA MANUAL DEL ARCHIVO NERRO ? (S/N)')
        READ(6,200)NS
        IF(NS.NE.ISN)GO TO 403
        WRITE(1,105)
105     FORMAT(//,5X,'INGRESE LOS DATOS DE LA SIGUIENTE MANERA :',//,
       6 'ER    ','SIGNIFICADO DEL CODIGO DE ERROR',
       6 44('.'))
        READ(5'1)NREG
        NREG=NREG+2
402     CONTINUE
        READ(6,120,END=99)NERRO
120     FORMAT(80A1)
        WRITE(5'NREG)NERRO
        NREG=NREG+1
        GO TO 402
99      NREG=NREG-2
        WRITE(5'1)NREG
403     CONTINUE
        WRITE(1,107)
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      PROGRAMA : ERROR
 5:C
 6:C      PROGRAMA QUE A PARTIR DE UN VALOR DE ERROR
 7:C      ESCRIBE EN PANTALLA EL SIGNIFICADO DE ESE
 8:C      CODIGO DE ERROR.
 9:C
10:       DEFINE FILE 5(100,160,U,J5)
11:       DIMENSION L80(80),L5(5),LB(80)
12:       DATA IBLK/' '/
13:       DATA ISN/'S'/
14:       DO 33 I1=1,80
15:       LB(I1)=IBLK
16:33     CONTINUE
17:41     WRITE(1,100)
18:100    FORMAT(//,5X,'INGRESE EL CODIGO DE ERROR (2 DIGITOS)',//,
19:      &'DD')
20:       READ(6,200)NERR
21:200    FORMAT(I2)
22:       CALL HRNTF(NERR,L5,1)
23:       CALL NTAAC(L5,1,5)
24:       READ(5'1)NREG
25:       NREG=NREG+1
26:       DO 31 J=2,NREG
27:       READ(5'J)L80
28:       JK=J
29:       DO 30 I=1,2
30:       KLM=I+3
31:       IF(L5(KLM).NE.L80(I))GO TO 31
32:30     CONTINUE
33:       GO TO 44
34:31     CONTINUE
35:       GO TO 46
36:44     WRITE(1,108)L80
37:108    FORMAT(80A1)
38:       JK=JK+1
39:       READ(5'JK)L80
40:       DO 32 II=1,80
41:       IF(L80(II).NE.LB(II))GO TO 44
42:32     CONTINUE
43:43     WRITE(1,109)
44:109    FORMAT(//,5X,'FIN DEL TRABAJO? (S/N)')
45:       READ(6,201)NS
46:201    FORMAT(A1)
47:       IF(NS.NE.ISN)GO TO 41
48:       GO TO 49
49:46     WRITE(1,111)
50:111    FORMAT(//,5X,'NO HAY ESE TIPO DE CODIGO DE ERROR')
51:       GO TO 43
52:49     CALL EXIT
53:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      PROGRAMA : LASCI
 5:C
 6:C      PROGRAMA PARA CARGAR LA TABLA CON CODIGOS ASCII.
 7:C
 8:       DEFINE FILE 3(250,6,U,J3)
 9:       DIMENSION LCD(2)
10:       DATA NS/'S'/
11:       WRITE(1,20)
12:20     FORMAT(//,5X,'( CARGA INICIAL ? (S/N) )')
13:       READ(6,80)IND
14:80     FORMAT(1A1)
15:       IF(IND.NE.NS)GO TO 40
16:       IP=1
17:       WRITE(3'1)IP
18:40     READ(3'1)IP
19:       WRITE(1,21)IP
20:21     FORMAT(//,5X,I6,' REGISTROS GRABADOS',/)
21:       WRITE(1,22)
22:22     FORMAT(/,5X,'INGRESE LA INFORMACION',//,'C CD')
23:41     READ(6,10,END=99)NC,LCD
24:10     FORMAT(1A1,1X,2A1)
25:       IP=IP+1
26:       WRITE(3'IP)NC,LCD
27:       GO TO 41
28:99     WRITE(1,21)IP
29:       WRITE(3'1)IP
30:       CALL EXIT
31:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      PROGRAMA : TANEM
 5:C
 6:C *** PROGRAMA PARA CARGAR UN ARCHIVO QUE CONTENGA
 7:C *** EL CODIGO NMOTECNICO Y TODOS LOS HEXADECIMALES
 8:C *** CORRESPONDIENTES A LOS DIFERENTES TIPOS DE DI-
 9:C *** RECCIONAMIENTOS.
10:C
11:       DEFINE FILE 1(400,32,U,J1)
12:       DIMENSION L(16)
13:       DATA ISN/'S'/
14:       DATA IBLK/' '/
15:       WRITE(1,1550)
16:1550   FORMAT(//,5X,'NOMENCLATURA : ',/,5X,14('='),//,
17:      6 5X,'NMON = CODIGO NMONICO',/,5X,'D1 = INMEDIATO',/,
18:      6 5X,'D2 = DIRECTO',/,5X,'D3 = INDEXADO',/,5X,
19:      6 'D4 = EXTENDIDO',/,5X,'D5 = IMPLICITO',/,5X,
20:      6 'D6 = RELATIVO')
21:       WRITE(1,100)
22:100    FORMAT(//,5X,'INICIALIZACION DEL ARCHIVO ? (S/N)')
23:       READ(6,110)NS
24:110    FORMAT(1A1)
25:       IF(NS.NE.ISN)GO TO 400
26:       DO 301 II=1,16
27:301    L(II)=IBLK
28:       JJ=0
29:       WRITE(1'1)JJ
30:       DO 300 I=2,200
31:300    WRITE(1'I)L
32:400    WRITE(1,120)
33:120    FORMAT(//,5X,'LECTURA DE ARCHIVO ? (S/N)')
34:       READ(6,110)NS
35:       IF(NS.NE.ISN)GO TO 410
36:       READ(1'1)NREG
37:       IF(NREG.EQ.0)GO TO 410
38:       WRITE(1,130)NREG
39:130    FORMAT(//,5X,I3,'  REGISTROS CON INFORMACION VALIDA',/)
40:       WRITE(1,200)
41:200    FORMAT(//,9X,'NMON D1 D2 D3 D4 D5 D6',/)
42:       NREG=NREG+1
43:       DO 302 J=2,NREG
44:       READ(1'J)L
45:       WRITE(1,140)J,L
46:140    FORMAT(5X,I3,1X,4A1,6(1X,2A1))
47:302    CONTINUE
48:       NREG=NREG-1
49:       WRITE(1'1)NREG
50:410    WRITE(1,150)
51:150    FORMAT(//,5X,'CARGA MANUAL ? (S/N)')
52:       READ(6,110)NS
53:       IF(NS.NE.ISN)GO TO 430
54:       WRITE(1,160)
55:160    FORMAT(//,5X,'INGRESE LOS DATOS ASI : ',//,
56:      6 'NMON D1 D2 D3 D4 D5 D6')
57:       READ(1'1)NREG
58:       NREG=NREG+2
59:420    READ(6,170,END=99)L
60:170    FORMAT(4A1,6(1X,2A1))
61:       WRITE(1'NREG)L
62:       NREG=NREG+1
63:       GO TO 420
```

```
 64:99     NREG=NREG-2
 65:       WRITE(1'1)NREG
 66:430    WRITE(1,230)
 67:230    FORMAT(//,5X,'DESEA MODIFICAR UN REGISTRO ? (S/N)')
 68:       READ(6,110)NS
 69:       IF(NS.NE.ISN)GO TO 440
 70:       WRITE(1,260)
 71:260    FORMAT(//,5X,'INGRESÈ EL PUNTERO DEL REGISTRO',/,'999')
 72:       READ(6,270)NREG
 73:270    FORMAT(I3)
 74:       WRITE(1,240)
 75:240    FORMAT(//,5X,'INGRESE EL REGISTRO MODIFICADO')
 76:       WRITE(1,160)
 77:       READ(6,170)L
 78:       WRITE(1'NREG)L
 79:440    WRITE(1,190)
 80:190    FORMAT(//,5X,'FIN DEL TRABAJO ? (S/N)')
 81:       READ(6,110)NS
 82:       IF(NS.NE.ISN)GO TO 400
 83:       CALL EXIT
 84:       END
-> LAST LINE
=>
```

```
   1:*EXTENDED PRECISION
   2:*TWO WORD INTEGERS
   3:C
   4:C      PROGRAMA : TEN
   5:C
   6:C      PROGRAMA PARA LISTAR LOS SIMBOLOS CARGADOS EN EL
   7:C      ARCHIVO ASCII.S
   8:C
   9:       DEFINE FILE 3(250,6,U,J3)
  10:       READ(3'1)NREC
  11:       DO 30 I=2,NREC
  12:       READ(3'I)NCH,N1,N2
  13:       WRITE(1,20)I,NCH,N1,N2
  14:20     FORMAT(I5,' : ',A1,3X,2A1)
  15:30     CONTINUE
  16:       READ(6,10)IND
  17:10     FORMAT(I11)
  18:       IF(IND.EQ.0)GO TO 40
  19:       READ(6,11)I,NCH,N1,N2
  20:11     FORMAT(I3,1X,I5,1X,I5,1X,I5)
  21:       WRITE(3'I)NCH,N1,N2
  22:40     CALL EXIT
  23:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      PROGRAMA : ALFCD
 5:C
 6:C      PROGRAMA QUE DEVUELVE EL CODIGO INTERNO DE LA MAQUINA DADO
 7:C      EL CARACTER ALFANUMERICO.
 8:C
 9:40     WRITE(1,20)
10:20     FORMAT(//,5X,'INGRESE EL CARACTER :',//,'X')
11:       READ(6,10)L
12:10     FORMAT(A1)
13:       WRITE(1,21)L,L
14:21     FORMAT(/,5X,A1,' = ',I12)
15:       GO TO 40
16:       END
17:       END
> LAST LINE
>
```

```
   1:*EXTENDED PRECISION
   2:*TWO WORD INTEGERS
   3:C
   4:C      PROGRAMA : CGPR
   5:C
   6:C      PROGRAMA PARA NUMERAR LAS LINEAS DEL PROGRAMA Y LIMPIAR
   7:C      EL CODIGO DE MAQUINA GENERADO EN UN ENSAMBLAJE ANTERIOR.
   8:C
   9:       SUBROUTINE CGPR
  10:       DIMENSION L(80)
  11:       DATA IBLK/' '/
  12:       READ(4'1)IP
  13:       DO 300 I=2,IP
  14:       J=I-1
  15:       READ(4'I)L
  16:       DO 301 KL=1,24
  17: 301   L(KL)=IBLK
  18:       CALL HRNTF(J,L,6)
  19:       CALL NTAAC(L,6,10)
  20:       WRITE(4'I)L
  21:300    CONTINUE
  22:       RETURN
  23:       END
  24:300    CONTINUE
  25:       RETURN
  26:       END
-> LAST LINE
=>
```

```
  1:*EXTENDED PRECISION
  2:*TWO WORD INTEGERS
  3:C
  4:C      SUBRUTINA : VDF
  5:C
  6:C *** SUBRUTINA QUE ANALIZA SI SE TIENE UNA DIRECTIVA
  7:C *** (PSEUDO-OPERACION) VALIDA (NTY.NE.0).
  8:C
  9:       SUBROUTINE VDF(L80,NTY)
 10:       INTEGER L3(3),L80(80)
 11:       DATA IBLK/' '/
 12:       NTY=0
 13:       DO 300 I=35,37
 14:       IF(L80(I).NE.IBLK)GO TO 400
 15:300    CONTINUE
 16:       NC=1
 17:       DO 301 I=32,34
 18:       L3(NC)=L80(I)
 19:301    NC=NC+1
 20:       CALL DIR(L3,NCD)
 21:       IF(NCD.EQ.0)GO TO 400
 22:       IF(NCD.EQ.5.OR.NCD.EQ.1)GO TO 40
 23:       GO TO 410
 24:40     DO 302 K=25,31
 25:       IF(L80(K).EQ.IBLK)GO TO 302
 26:       RETURN
 27:302    CONTINUE
 28:410    NTY=NCD
 29:400    RETURN
 30:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : PERR
 5:C
 6:C      SUBRUTINA QUE ESCRIBE EN LAS LINEAS DEL PROGRAMA LOS MEN -
 7:C      SAJES DE ERROR CORRESPONDIENTES, DADO EL NUMERO DEL ERROR.
 8:C
 9:       SUBROUTINE PERR(L80,NER,IT)
10:       DIMENSION L80(80),L5(5)
11:       DATA NAST,NERN/'*','E'/
12:       L80(1)=NAST
13:       L80(2)=NERN
14:       L80(5)=NAST
15:       CALL HRNTF(NER,L5,1)
16:       CALL NTAAC(L5,1,5)
17:       L80(3)=L5(4)
18:       L80(4)=L5(5)
19:       WRITE(4'IT)L80
20:       RETURN
21:       END
> LAST LINE
>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : SRORG
 5:C
 6:C      SUBRUTINA QUE DA EL TRATAMIENTO ADECUADO A UNA LINEA DEL PROGRA -
 7:C      MA QUE CONTENGA UN "ORG".
 8:C
 9:       SUBROUTINE SRORG(L80,NDEQ4,NER,IT)
10:       DIMENSION L80(80),L18(18),L4MC(4)
11:       NMAX=65535
12:       NER=0
13:       CALL MOVER(L80,38,55,L18,1)
14:       CALL NTYPE(L18,NC,NDEQ4,NCM,NPP)
15:       IF(NC.NE.0.AND.NCM.NE.2)GO TO 400
16:       GO TO 402
17:400    IF(NDEQ4.GT.NMAX)GO TO 401
18:       NER=1
19:       CALL DTH(NDEQ4,L4MC)
20:       CALL NTAAC(L4MC,1,4)
21:       CALL MOVER(L4MC,1,4,L80,12)
22:       WRITE(4'IT)L80
23:       RETURN
24:401    NER=3
25:       CALL PERR(L80,10,IT)
26:402    RETURN
27:       END
> LAST LINE
>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : SREQU
 5:C
 6:C      SUBRUTINA QUE DA EL TRATAMIENTO ADECUADO A UNA LINEA DE PRO -
 7:C      GRAMA QUE CONTIENE UN "EQU".
 8:C
 9:       SUBROUTINE SREQU(L80,NQ4,NER,IT,NSH)
10:       DIMENSION L80(80),LD5(5),L18(18),L4MC(4)
11:       DIMENSION LNET(5)
12:       NER=0
13:       CALL MOVER(L80,38,55,L18,1)
14:       CALL NTYPE(L18,NC,NDEQ4,NCM,NPP)
15:       IF(NC.EQ.0)CALL PERR(L80,10,IT)
16:       CALL MOVER (L80,25,29,LD5,1)
17:       CALL MOVER(L80,25,42,L18,1)
18:       CALL LABEL(L18,NE,NDEC,LNET)
19:       IF(NE.NE.1)GO TO 400
20:       NER=1
21:       CALL DTH(NDEQ4,L4MC)
22:       CALL NTAAC(L4MC,1,4)
23:       CALL MOVER(L4MC,1,4,L80,17)
24:       WRITE(4'IT)L80
25:       IF(NSH.EQ.2)RETURN
26:       READ(2'1)NREC
27:       NREC=NREC+1
28:       IF(NREC.LE.520)GO TO 42
29:       CALL PERR(L80,13,IT)
30:       RETURN
31:42     WRITE(2'1)NREC
32:       WRITE(2'NREC)LD5,NDEQ4
33:       RETURN
34:400    NER=2
35:       CALL PERR(L80,9,IT)
36:       RETURN
37:       END
> LAST LINE
>
```

```
*EXTENDED PRECISION
*TWO WORD INTEGERS
C
C       SUBRUTINA : SRRMB
C
C       SUBRUTINA QUE DA EL TRATAMIENTO CORRESPONDIENTE A UNA LINEA DE
C       PROGRAMA QUE CONTEGA UN "RMB".
C
        SUBROUTINE SRRMB(L80,NQ4,NER,IT,NSH)
        DIMENSION L80(80),L18(18),L4MC(4),LD5(5),LNET(5)
        NER=0
        N64K=65535
        CALL MOVER(L80,38,55,L18,1)
        CALL NTYPE(L18,NC,NDEQ4,NCM,NPP)
        IF(NC.EQ.0)RETURN
        IF((NQ4+NDEQ4-1).LE.N64K)GO TO 40
        NER=3
        CALL PERR(L80,10,IT)
        RETURN
40      CALL MOVER(L80,25,29,LD5,1)
        CALL MOVER(L80,25,42,L18,1)
        CALL LABEL(L18,NE,NDEC,LNET)
        IF(NE.EQ.1)GO TO 41
        NER=2
        CALL PERR(L80,9,IT)
        RETURN
41      NER=1
        CALL DTH(NQ4,L4MC)
        CALL NTAAC(L4MC,1,4)
        CALL MOVER(L4MC,1,4,L80,12)
        CALL DTH(NDEQ4,L4MC)
        CALL NTAAC(L4MC,1,4)
        CALL MOVER(L4MC,1,4,L80,17)
        WRITE(4'IT)L80
        IF(NSH.EQ.2)GO TO 4000
        READ(2'1)NREC
        NREC=NREC+1
        IF(NREC.LE.520)GO TO 42
        CALL PERR(L80,13,IT)
        GO TO 4000
42      WRITE(2'NREC)LD5,NQ4
        WRITE(2'1)NREC
4000    NQ4=NQ4+NDEQ4
        RETURN
        END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : SRFCC
 5:C
 6:C      SUBRUTINA QUE DA EL TRATAMIENTO NECESARIO A UNA LINEA DE PROGRAMA
 7:C      QUE CONTENGA UN "FCC".
 8:C
 9:       SUBROUTINE SRFCC(L80,NQ4,NER,IT,NSH)
10:       DIMENSION L80(80),LD5(5),L18(18),L4MC(4),LNET(5)
11:       NER=0
12:       NCHAR=L80(38)
13:       READ(3'1)NRA
14:       DO 30 I=2,NRA
15:       READ(3'I)NC,LCD1,LCD2
16:       IF(NC.NE.NCHAR)GO TO 30
17:       CALL MOVER (L80,25,29,LD5,1)
18:       CALL MOVER(L80,25,42,L18,1)
19:       CALL LABEL(L18,NE,NDEC,LNET)
20:       IF(NE.NE.1)GO TO 400
21:       NER=1
22:       CALL DTH(NQ4,L4MC)
23:       CALL NTAAC(L4MC,1,4)
24:       CALL MOVER(L4MC,1,4,L80,12)
25:       L80(17)=LCD1
26:       L80(18)=LCD2
27:       WRITE(4'IT)L80
28:       IF(NSH.EQ.2)GO TO 4000
29:       READ(2'1)NREC
30:       NREC=NREC+1
31:       IF(NREC.LE.520)GO TO 42
32:       CALL PERR(L80,13,IT)
33:       GO TO 4000
34:42     WRITE(2'1)NREC
35:       WRITE(2'NREC)LD5,NQ4
36:4000   NQ4=NQ4+1
37:       RETURN
38:30     CONTINUE
39:       NER=3
40:       RETURN
41:400    NER=2
42:       CALL PERR(L80,9,IT)
43:       RETURN
44:       END
> LAST LINE
>
```

```
  1:*EXTENDED PRECISION
  2:*TWO WORD INTEGERS
  3:C
  4:C      SUBRUTINA : SRFCB
  5:C
  6:C      SUBRUTINA PARA DAR UN TRATAMIENTO ADECUADO A UNA LINEA DE
  7:C      PROGRAMA QUE CONTENGA UN "FCB".
  8:C
  9:       SUBROUTINE SRFCB(L80,NQ4,NER,IT,NSH)
 10:       DIMENSION L80(80),LD5(5),L18(18),L4MC(4),LNET(5)
 11:       NER=0
 12:       CALL MOVER(L80,38,55,L18,1)
 13:       CALL NTYPE(L18,NC,NDEQ4,NCM,NPP)
 14:       IF(NC.EQ.0)CALL PERR(L80,10,IT)
 15:       IF(NDEQ4.GT.255)GO TO 401
 16:       CALL MOVER (L80,25,29,LD5,1)
 17:       CALL MOVER(L80,25,42,L18,1)
 18:       CALL LABEL(L18,NE,NDEC,LNET)
 19:       IF(NE.NE.1)GO TO 400
 20:       NER=1
 21:       CALL DTH(NQ4,L4MC)
 22:       CALL NTAAC(L4MC,1,4)
 23:       CALL MOVER(L4MC,1,4,L80,12)
 24:       CALL DTH(NDEQ4,L4MC)
 25:       CALL NTAAC(L4MC,1,4)
 26:       CALL MOVER(L4MC,3,4,L80,17)
 27:       WRITE(4'IT)L80
 28:       IF(NSH.EQ.2)GO TO 4000
 29:       READ(2'1)NREC
 30:       NREC=NREC+1
 31:       IF(NREC.LE.520)GO TO 42
 32:       CALL PERR(L80,13,IT)
 33:       GO TO 4000
 34:42     WRITE(2'1)NREC
 35:       WRITE(2'NREC)LD5,NQ4
 36:4000   NQ4=NQ4+1
 37:       RETURN
 38:400    NER=2
 39:       CALL PERR(L80,9,IT)
 40:       RETURN
 41:401    NER=3
 42:       NQ4=NQ4+1
 43:       CALL PERR(L80,10,IT)
 44:       RETURN
 45:       END
> LAST LINE
>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : CMP
 5:C
 6:C      SUBRUTINA QUE COMPARA SI LOS ARREGLOS L1 Y L2 CONTIENEN LA
 7:C      MISMA INFORMACION.
 8:C
 9:       SUBROUTINE CMP(L1,L2,NM)
10:       INTEGER L1(3),L2(3)
11:       NM=0
12:       DO 30 I=1,3
13:       IF(L1(I).NE.L2(I))GO TO 40
14:30     CONTINUE
15:       NM=1
16:40     RETURN
17:       END
* LAST LINE
*
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : MOVER
 5:C
 6:C      SUBRUTINA PARA MOVER CIERTOS ELEMENTOS DE UN ARREGLO A
 7:C      CIERTOS ELEMENTOS DE OTRO ARREGLO.
 8:C
 9:       SUBROUTINE MOVER(LA,NI,NF,LB,NIAUX)
10:       INTEGER LA(80),LB(80)
11:       NII=NIAUX
12:       DO 30 I=NI,NF
13:       LB(NII)=LA(I)
14:       NII=NII+1
15:30     CONTINUE
16:       RETURN
17:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : VNMOP
 5:C
 6:C      SUBRUTINA QUE VERIFICA SI EXISTE EN ESA LINEA UN CODIGO
 7:C      NMONICO VALIDO.
 8:C
 9:       SUBROUTINE VNMOP(L80,LCOD,NER)
10:       DIMENSION L80(80),LCOD(16)
11:       NER=0
12:       READ(1'1)NREC
13:       DO 300 I=2,NREC
14:       READ(1'I)LCOD
15:       IF(L80(32).NE.LCOD(1).OR.L80(33).NE.LCOD(2).OR.
16:      6   L80(34).NE.LCOD(3).OR.L80(35).NE.LCOD(4))GO TO 300
17:       NER=1
18:       RETURN
19:300    CONTINUE
20:       RETURN
21:       END
> LAST LINE
>
```

```
*EXTENDED PRECISION
*TWO WORD INTEGERS
C
C      SUBRUTINA : NTYPE
C
C *** SUBRUTINA QUE RECIBE EL OPERANDO DE HASTA 17 CARAC-
C *** TERES Y DEVUELVE UN CODIGO NC QUE IDENTIFICA EL TI-
C *** PO DE NUMERO ASI:
C *** NC=0 ... NO NUMERO
C *** NC=1 ... BINARIO
C *** NC=2 ... OCTAL
C *** NC=3 ... DECIMAL
C *** NC=4 ... HEXADECIMAL
C *** ADEMAS ENVIA UN EQUIVALENTE DECIMAL DEL NUMERO (NDEQ).
C
      SUBROUTINE NTYPE(LSV,NC,NDEQ,NCM,NPP)
      DIMENSION L(18),LN(16),LSV(18)
      DATA LN(1)-(10)/'1','2','3','4','5','6','7','8','9','A'/
      DATA LN(11)-(16)/'B','C','D','E','F','0'/
      DATA IBLK/' '/
      DATA ICOM/','/
      DATA ISUC,IARR,IPOR/'$',2084585536,'%'/
      DO 370 I=1,18
370   L(I)=LSV(I)
      NC=0
      NCM=0
      NPP=0
      NDEQ=0
      IF(L(1).EQ.ISUC)GO TO 415
      IF(L(1).EQ.IARR)GO TO 416
      IF(L(1).EQ.IPOR)GO TO 417
      DO 314 K=1,5
      DO 305 I=1,9
      IF(L(K).EQ.LN(I))GO TO 434
      IF(L(K).EQ.LN(16))GO TO 434
      GO TO 305
434   IF(L(K).NE.LN(16))L(K)=I
      IF(L(K).EQ.LN(16))L(K)=0
      GO TO 314
305   CONTINUE
432   IF(L(K).NE.IBLK.AND.L(K).NE.ICOM)GO TO 901
      IF(L(K).EQ.IBLK)NCM=1
      IF(L(K).EQ.ICOM)NCM=2
      GO TO 433
314   CONTINUE
      K=6
      GO TO 432
433   ND=K-1
      IF(ND.LE.0)GO TO 901
      NC=3
      NDEQ=L(1)
      IF(ND.EQ.1)GO TO 900
      DO 313 K=2,ND
313   NDEQ=NDEQ*10+L(K)
      GO TO 900
415   DO 304 I=2,5
      DO 306 J=1,16
      IF(L(I).EQ.LN(J))GO TO 422
      GO TO 306
422   L(I)=J
      IF(L(I).EQ.16)L(I)=0
      GO TO 304
306   CONTINUE
```

```
 64:424    IF(L(I).NE.IBLK.AND.L(I).NE.ICOM)GO TO 901
 65:       IF(L(I).EQ.IBLK)NCM=1
 66:       IF(L(I).EQ.ICOM)NCM=2
 67:       GO TO 407
 68:304    CONTINUE
 69:       I=6
 70:       GO TO 424
 71:407    IF(I.LE.2)GO TO 901
 72:       ND=I-1
 73:       NC=4
 74:       NDEQ=L(2)
 75:       IF(ND.EQ.2)GO TO 900
 76:       DO 307 I=3,ND
 77:307    NDEQ=NDEQ*16+L(I)
 78:       GO TO 900
 79:416    CONTINUE
 80:       DO 302 I2=2,7
 81:       DO 309 J2=1,7
 82:       IF(L(I2).EQ.LN(J2))GO TO 426
 83:       IF(L(I2).EQ.LN(16))GO TO 426
 84:       GO TO 309
 85:426    IF(L(I2).NE.LN(16))L(I2)=J2
 86:       IF(L(I2).EQ.LN(16))L(I2)=0
 87:       GO TO 302
 88:309    CONTINUE
 89:428    IF(L(I2).NE.IBLK.AND.L(I2).NE.ICOM)GO TO 901
 90:       IF(L(I2).EQ.IBLK)NCM=1
 91:       IF(L(I2).EQ.ICOM)NCM=2
 92:       GO TO 406
 93:302    CONTINUE
 94:       I2=8
 95:       GO TO 428
 96:406    IF(I2.LE.2)GO TO 901
 97:       ND=I2-1
 98:       NC=2
 99:       NDEQ=L(2)
100:       IF(ND.EQ.2)GO TO 900
101:       DO 310 I2=3,ND
102:310    NDEQ=NDEQ*8+L(I2)
103:       GO TO 900
104:417    CONTINUE
105:       DO 301 I1=2,17
106:       IF(L(I1).EQ.LN(16))GO TO 430
107:       IF(L(I1).EQ.LN(1))GO TO 427
108:       GO TO 429
109:427    L(I1)=1
110:       GO TO 301
111:430    L(I1)=0
112:       GO TO 301
113:429    IF(L(I1).NE.IBLK.AND.L(I1).NE.ICOM)GO TO 901
114:       IF(L(I1).EQ.IBLK)NCM=1
115:       IF(L(I1).EQ.ICOM)NCM=2
116:       GO TO 405
117:301    CONTINUE
118:       I1=18
119:       GO TO 429
120:405    IF(I1.LE.2)GO TO 901
121:       ND=I1-1
122:       NC=1
123:       NDEQ=L(2)
124:       IF(ND.EQ.2)GO TO 900
125:       DO 311 I1=3,ND
126:311    NDEQ=NDEQ*2+L(I1)
127:       GO TO 900
128:901    NC=0
129:       NDEQ=0
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : LABEL
 5:C
 6:C      SUBRUTINA QUE ANALIZA EL OPERANDO Y DETERMINA SI ES O NO UNA
 7:C      ETIQUETA, Y EN CASO AFIRMATIVO, QUE TIPO DE ETIQUETA ES.
 8:C
 9:       SUBROUTINE LABEL(LA,NE,NDEC,LNET)
10:       DIMENSION LE(18),LA(18),LNET(5)
11:       DIMENSION LN(36)
12:       DATA LN(1)-(10)/'1','2','3','4','5','6','7','8','9','0'/
13:       DATA LN(11)-(20)/'A','B','C','D','E','F','G','H','I','J'/
14:       DATA LN(21)-(30)/'K','L','M','N','O','P','Q','R','S','T'/
15:       DATA LN(31)-(36)/'U','V','W','Z','Y','X'/
16:       DATA IBLK,IMAS,IMEN/' ','+','-'/
17:       DO 355 I=1,5
18:355    LNET(I)=IBLK
19:       DO 333 I=1,18
20:333    LE(I)=LA(I)
21:       NE=0
22:       NDEC=0
23:       DO 300 K=11,35
24:       IF(LE(1).EQ.LN(K))GO TO 401
25:300    CONTINUE
26:       GO TO 901
27:401    DO 303 I1=2,5
28:       DO 304 K=1,36
29:       IF(LE(I1).EQ.LN(K))GO TO 303
30:304    CONTINUE
31:       GO TO 407
32:303    CONTINUE
33:       I1=6
34:407    CONTINUE
35:       I1M1=I1-1
36:       DO 350 KL=1,I1M1
37:350    LNET(KL)=LE(KL)
38:       IF(LE(I1).EQ.IBLK)GO TO 408
39:       IF(LE(I1).NE.IMAS.AND.LE(I1).NE.IMEN)GO TO 901
40:       IF(LE(I1).EQ.IMAS)NE=2
41:       IF(LE(I1).EQ.IMEN)NE=3
42:       I2=I1+1
43:       I3=I2+4
44:       DO 305 I1=I2,I3
45:       DO 306 K=1,10
46:       IF(LE(I1).EQ.LN(K))GO TO 305
47:306    CONTINUE
48:       GO TO 400
49:305    CONTINUE
50:       I1=I3+1
51:400    CONTINUE
52:       IF(LE(I1).NE.IBLK)GO TO 901
53:       IF(LE(I1).EQ.IBLK.AND.I1.EQ.I2)GO TO 901
54:       I1=I1-1
55:       CALL ATNAC(LE,I2,I1)
56:       NDEC=LE(I2)
57:       IF((I1-I2+1).EQ.1)RETURN
58:       I2=I2+1
59:       DO 301 I3=I2,I1
60:       NDEC=NDEC*10+LE(I3)
61:301    CONTINUE
62:       IF(NDEC.LT.65535)RETURN
63:       NE=4
```

```
1:*EXTENDED PRECISION
2:*TWO WORD INTEGERS
3:C
4:C      SUBRUTINA : INMED
5:C
6:C      SUBRUTINA QUE SE ENCARGA DE CONSIDERAR SI LA LINEA CONTIENE
7:C      UNA INSTRUCCION CON DIRECCIONAMIENTO INMEDIATO.
8:C
9:       SUBROUTINE INMED(LSV,NX)
10:      DIMENSION LSV(18),L(18),LY(4),LTY(4,4)
11:      DATA LY/'S','#','N',' '/
12:      NX=0
13:      DO 300 I=1,4
14:      DO 300 J=1,4
15:300   LTY(I,J)=0
16:      LTY(1,1)=2
17:      LTY(2,2)=3
18:      LTY(3,3)=4
19:      LTY(4,4)=1
20:      DO 302 J2=1,17
21:      J3=J2+1
22:      L(J2)=LSV(J3)
23:302   CONTINUE
24:      CALL NTYPE(L,NC,NDEQ,NCM,NPP)
25:      KY=LTY(1,1)
26:      IF(LSV(1).NE.LY(KY))GO TO 901
27:      IF(NC.EQ.0)GO TO 901
28:      IF(NCM.NE.1)GO TO 901
29:      NX=1
30:901   RETURN
31:      END
LAST LINE
```

```
   1:*EXTENDED PRECISION
   2:*TWO WORD INTEGERS
   3:C
   4:C      SUBRUTINA : RELAT
   5:C
   6:C      SUBRUTINA QUE SE ENCARGA DE RECONOCER EL DIRECCIONAMIENTO
   7:C      RELATIVO
   8:C
   9:       SUBROUTINE RELAT(L80,LR,NE,NX)
  10:       DIMENSION LR(16),L80(80)
  11:       INTEGER*4 JMP(3),JSR(3),LCMP(3)
  12:       DATA JMP/'J','M','P'/,JSR/'J','S','R'/
  13:       DATA IBLK/' '/
  14:       NM=0
  15:       IF(L80(35).NE.IBLK)GO TO 480
  16:       CALL MOVER(L80,32,34,LCMP,1)
  17:       CALL CMP(LCMP,JMP,NM)
  18:       IF(NM.EQ.1)GO TO 480
  19:       CALL CMP(LCMP,JSR,NM)
  20:480    CONTINUE
  21:       IF(NM.EQ.0)GO TO 481
  22:       NX=4
  23:       RETURN
  24:481    NX=0
  25:       IF(NE.EQ.1.OR.NE.EQ.2.OR.NE.EQ.3)GO TO 40
  26:       RETURN
  27:40     IF(LR(15).EQ.IBLK)RETURN
  28:       IF(LR(16).EQ.IBLK)RETURN
  29:       NX=6
  30:       RETURN
  31:       END
-> LAST LINE
=>
```

```
    1:*EXTENDED PRECISION
    2:*TWO WORD INTEGERS
    3:C
    4:C     SUBRUTINA : INDEX
    5:C
    6:C     SUBRUTINA QUE IDENTIFICA EL DIRECCIONAMIENTO INDEXADO.
    7:C
    8:      SUBROUTINE INDEX(LSV,NC,NDEQ,NCM,NPP,NX)
    9:      DIMENSION LSV(18),LZ(4),LTX(4,4)
   10:      DATA LZ/'S','N','X',' '/
   11:      DO 300 I=1,4
   12:      DO 300 J=1,4
   13:300   LTX(I,J)=0
   14:      LTX(1,1)=2
   15:      LTX(1,2)=3
   16:      LTX(2,2)=3
   17:      LTX(3,3)=4
   18:      LTX(4,4)=1
   19:      NSW=0
   20:      I1=1
   21:410   CONTINUE
   22:      DO 31 I2=1,3
   23:      IF(LTX(I1,I2).EQ.0)GO TO 31
   24:      IF(NSW.NE.0)GO TO 420
   25:      NSW=1
   26:      IF(NC.NE.0.AND.NCM.EQ.2)GO TO 400
   27:401   NPP=1
   28:      GO TO 31
   29:400   IF(NDEQ.GT.65535)GO TO 401
   30:      KZ=LTX(I1,I2)
   31:      I1=KZ
   32:      GO TO 410
   33:420   KZ=LTX(I1,I2)
   34:      IF(LSV(NPP).EQ.LZ(KZ))GO TO 430
   35:      GO TO 31
   36:430   I1=KZ
   37:      NPP=NPP+1
   38:      GO TO 410
   39:31    CONTINUE
   40:      I3=4
   41:      IF(LTX(I1,I3).EQ.1)GO TO 900
   42:      NX=0
   43:      RETURN
   44:900   NX=3
   45:      RETURN
   46:      END
--> LAST LINE
E=>
```

```
    1:*EXTENDED PRECISION
    2:*TWO WORD INTEGERS
    3:C
    4:C      SUBRUTINA : DIREC
    5:C
    6:C      SUBRUTINA QUE SE ENCARGA DE DIFERENCIAR ENTRE DIRECCIONAMIENTO
    7:C      DIRECTO Y EXTENDIDO.
    8:C
    9:       SUBROUTINE DIREC(NDEN,NE,NC,NX)
   10:       NX=0
   11:       IF(NC.EQ.0)GO TO 40
   12:       IF(NDEN.LT.255)GO TO 41
   13:       NX=4
   14:       RETURN
   15:41     NX=2
   16:       RETURN
   17:40     IF(NE.EQ.0.OR.NE.EQ.4)GO TO 42
   18:       NX=2
   19:42     RETURN
   20:       END
--> LAST LINE
E=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : IMPLI
 5:C
 6:C      SUBRUTINA QUE IDENTIFICA EL DIRECCIONAMIENTO IMPLICITO.
 7:C
 8:       SUBROUTINE IMPLI(LSV,NC,LR,NX)
 9:       DIMENSION LSV(18),LR(16)
10:       DATA IBLK/' '/
11:       NX=0
12:       IF(NC.NE.0)RETURN
13:       IF(LSV(1).NE.IBLK)RETURN
14:       IF(LR(15).NE.IBLK.OR.LR(16).NE.IBLK)RETURN
15:       NX=5
16:       RETURN
17:       END
 LAST LINE
```

```
  1:*EXTENDED PRECISION
  2:*TWO WORD INTEGERS
  3:C
  4:C      SUBRUTINA : LCAR
  5:C
  6:C      SUBRUTINA QUE CARGA EL CODIGO NMOTECNICO EN EL ARREGLO LCODE,
  7:C      DE ACUERDO AL TIPO DE DIRECCIONAMIENTO.
  8:C
  9:       SUBROUTINE LCAR(LCODE,LR,NX)
 10:       DIMENSION LR(16),LCODE(6)
 11:       LR1=2*NX+3
 12:       LR2=LR1+1
 13:       LCODE(1)=LR(LR1)
 14:       LCODE(2)=LR(LR2)
 15:       RETURN
 16:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : COD1
 5:C
 6:C      SUBRUTINA QUE GENERA EL CODIGO DEL OPERANDO Y LA DIRECCION
 7:C      HEXADECIMAL CORRESPONDIENTE PARA EL DIRECCIONAMIENTO IN  -
 8:C      MEDIATO.
 9:C
10:       SUBROUTINE COD1(L80,NQ4,IT)
11:       DIMENSION L80(80),L4MC(4),L4HEX(4),L18(18)
12:       INTEGER*4 CPX(3),LDX(3),LDS(3),LCOMP(3)
13:       DATA CPX/'C','P','X'/,LDX/'L','D','X'/,LDS/'L','D','S'/
14:       DATA IBLK/' '/
15:       NM=0
16:       IF(L80(35).NE.IBLK)GO TO 41
17:       CALL MOVER(L80,32,34,LCOMP,1)
18:       CALL CMP(LCOMP,CPX,NM)
19:       IF(NM.EQ.1)GO TO 41
20:       CALL CMP(LCOMP,LDX,NM)
21:       IF(NM.EQ.1)GO TO 41
22:       CALL CMP(LCOMP,LDS,NM)
23:41     CALL MOVER(L80,39,56,L18,1)
24:       CALL NTYPE(L18,NC,NDEQ,NCM,NPP)
25:       IF(NM.EQ.1)GO TO 440
26:       IF(NDEQ.LE.255)GO TO 40
27:       CALL PERR(L80,10,IT)
28:       GO TO 45
29:40     CALL DTH(NDEQ,L4HEX)
30:       CALL NTAAC(L4HEX,1,4)
31:       L80(20)=L4HEX(3)
32:       L80(21)=L4HEX(4)
33:45     CALL DTH(NQ4,L4MC)
34:       CALL NTAAC(L4MC,1,4)
35:       CALL MOVER(L4MC,1,4,L80,12)
36:       WRITE(4'IT)L80
37:       NQ4=NQ4+2
38:       RETURN
39:C
40:C      TRATAMIENTO ESPECIAL PARA CPX,LDX, Y LDS.
41:C
42:440    IF(NDEQ.LE.65535)GO TO 441
43:       CALL PERR(L80,10,IT)
44:       GO TO 442
45:441    CALL DTH(NDEQ,L4HEX)
46:       CALL NTAAC(L4HEX,1,4)
47:       CALL MOVER(L4HEX,1,4,L80,20)
48:442    CALL DTH(NQ4,L4MC)
49:       CALL NTAAC(L4MC,1,4)
50:       CALL MOVER(L4MC,1,4,L80,12)
51:       WRITE(4'IT)L80
52:       NQ4=NQ4+3
53:       RETURN
54:       END
> LAST LINE
>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : COD24
 5:C
 6:C      SUBRUTINA QUE GENERA EL CODIGO DEL OPERANDO Y DIRECCION EN MEMORI
 7:C      PARA EL CASO DE DIRECCIONAMIENTOS DIRECTO O EXTENDIDO.
 8:C
 9:       SUBROUTINE COD24(L80,NQ4,LCOD,NSH,IT)
10:       INTEGER JMP(3),JSR(3),LCOMP(3)
11:       DIMENSION L80(80),L4MC(4),L4HEX(4),L18(18),LD5(5),LCOD(16)
12:       DATA JMP/'J','M','P'/,JSR/'J','S','R'/
13:       DATA IBLK/' '/
14:       NM=0
15:       IF(L80(35).NE.IBLK)GO TO 480
16:       CALL MOVER(L80,32,34,LCOMP,1)
17:       CALL CMP(LCOMP,JMP,NM)
18:       IF(NM.EQ.1)GO TO 480
19:       CALL CMP(LCOMP,JSR,NM)
20:480    CONTINUE
21:       IF(L80(17).NE.IBLK.AND.L80(18).NE.IBLK)GO TO 481
22:       L80(17)=LCOD(11)
23:       L80(18)=LCOD(12)
24:       NM=1
25:481    CONTINUE
26:       CALL MOVER(L80,38,55,L18,1)
27:       CALL LABEL(L18,NE,NDEC,LD5)
28:       IF(NE.NE.0)GO TO 42
29:       CALL MOVER(L80,38,55,L18,1)
30:       CALL NTYPE(L18,NC,NDEQ,NCM,NPP)
31:       IF(NDEQ.LE.255.AND.NM.EQ.0)GO TO 40
32:       IF(NDEQ.LE.65535)GO TO 440
33:       CALL PERR(L80,10,IT)
34:       GO TO 45
35:40     CALL DTH(NDEQ,L4HEX)
36:       CALL NTAAC(L4HEX,1,4)
37:       L80(20)=L4HEX(3)
38:       L80(21)=L4HEX(4)
39:       GO TO 49
40:42     IF(NE.NE.4)GO TO 43
41:       CALL PERR(L80,2,IT)
42:       GO TO 49
43:43     CALL SCHLB(LD5,NLB,NB)
44:       IF(NB.EQ.0.OR.NB.EQ.1)GO TO 400
45:       CALL PERR(L80,8,IT)
46:       GO TO 49
47:400    IF(NB.NE.0)GO TO 44
48:       IF(NSH.EQ.1)GO TO 49
49:       CALL PERR(L80,8,IT)
50:       GO TO 49
51:44     IF(NE.NE.1)GO TO 46
52:       NDEQ=NLB
53:       IF(NDEQ.GT.255.OR.NM.EQ.1)GO TO 440
54:       GO TO 40
55:46     IF(NE.EQ.2)NDEQ=NLB+NDEC
56:       IF(NE.EQ.3)NDEQ=NLB-NDEC
57:       IF(NDEQ.GE.0)GO TO 41
58:       CALL PERR(L80,10,IT)
59:       GO TO 49
60:41     IF(NDEQ.GT.255.OR.NM.EQ:1)GO TO 440
61:       GO TO 40
62:49     CONTINUE
63:       CALL DTH(NQ4,L4MC)
```

```
 64:        CALL NTAAC(L4MC,1,4)
 65:        CALL MOVER(L4MC,1,4,L80,12)
 66:        WRITE(4'IT)L80
 67:45      NQ4=NQ4+2
 68:        RETURN
 69:440     CALL DTH(NDEQ,L4MC)
 70:        CALL NTAAC(L4MC,1,4)
 71:        CALL MOVER(L4MC,1,4,L80,20)
 72:        CALL DTH(NQ4,L4MC)
 73:        CALL NTAAC(L4MC,1,4)
 74:        CALL MOVER(L4MC,1,4,L80,12)
 75:        WRITE(4'IT)L80
 76:        NQ4=NQ4+3
 77:        RETURN
 78:        END
-> LAST LINE
=>
```

```
  1:*EXTENDED PRECISION
  2:*TWO WORD INTEGERS
  3:C
  4:C      SUBRUTINA : COD3
  5:C
  6:C      SUBRUTINA QUE GENERA EL CODIGO DEL OPERANDO Y DIRECCION EN MEMORI
  7:C      PARA EL CASO DE DIRECCIONAMIENTO INDEXADO.
  8:C
  9:       SUBROUTINE COD3(L80,NQ4,NSH,IT)
 10:       DIMENSION L80(80),L4MC(4),L18(18)
 11:       CALL MOVER(L80,38,55,L18,1)
 12:       CALL NTYPE(L18,NC,NDEQ,NCM,NPP)
 13:       IF(NDEQ.LE.255)GO TO 40
 14:       CALL PERR(L80,10,IT)
 15:       GO TO 41
 16:40     IF(NC.EQ.0)NDEQ=0
 17:       CALL DTH(NDEQ,L4MC)
 18:       CALL NTAAC(L4MC,1,4)
 19:       CALL MOVER(L4MC,3,4,L80,20)
 20:41     CALL DTH(NQ4,L4MC)
 21:       CALL NTAAC(L4MC,1,4)
 22:       CALL MOVER(L4MC,1,4,L80,12)
 23:       WRITE(4'IT)L80
 24:       NQ4=NQ4+2
 25:       RETURN
 26:       END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : COD5
 5:C
 6:C      SUBRUTINA PARA GENERAR LAS DIRECCIONES EN MEMORIA CORRES -
 7:C      PONDIENTES A ESA LINEA DEL PROGRAMA PARA EL CASO DE DIREC -
 8:C      CIONAMIENTO IMPLICITO.
 9:C
10:      SUBROUTINE COD5(L80,NQ4,NSH,IT)
11:      DIMENSION L80(80),L4MC(4)
12:      CALL DTH(NQ4,L4MC)
13:      CALL NTAAC(L4MC,1,4)
14:      CALL MOVER(L4MC,1,4,L80,12)
15:      WRITE(4'IT)L80
16:      NQ4=NQ4+1
17:      RETURN
18:      END
> LAST LINE
>
```

```
   1:*EXTENDED PRECISION
   2:*TWO WORD INTEGERS
   3:C
   4:C      SUBRUTINA : COD6
   5:C
   6:C      SUBRUTINA QUE GENERA EL CODIGO DEL OPERANDO Y LA DIRECCION
   7:C      EN MEMORIA PARA EL CASO DEL DIRECCIONAMIENTO RELATIVO.
   8:C
   9:       SUBROUTINE COD6(L80,NQ4,NSH,IT)
  10:       DIMENSION L80(80),L4MC(4),L18(18),LNET(5)
  11:C
  12:C      CALCULAR : # = D.D. - P.D.
  13:C                 DONDE  ---  #    : OFFSET
  14:C                             D.D. : DIRECCION DE DESTINO
  15:C                             P.D. : PROXIMA DIRECCION
  16:C
  17:       CALL MOVER(L80,38,55,L18,1)
  18:       CALL LABEL(L18,NE,NDEC,LNET)
  19:       IF(NE.EQ.1)GO TO 40
  20:       CALL PERR(L80,3,IT)
  21:       GO TO 49
  22:40     CALL SCHLB(LNET,NLB,NB)
  23:       IF(NB.EQ.0.OR.NB.EQ.1)GO TO 41
  24:       CALL PERR(L80,4,IT)
  25:       GO TO 49
  26:41     IF(NB.NE.0)GO TO 42
  27:       IF(NSH.EQ.1)GO TO 49
  28:       CALL PERR(L80,8,IT)
  29:       GO TO 49
  30:42     LPD=NQ4+2
  31:       LDD=NLB
  32:       NOFFS=LDD-LPD
  33:       IF(ABS(NOFFS).LE.255)GO TO 44
  34:       CALL PERR(L80,10,IT)
  35:       GO TO 49
  36:44     IF(NOFFS.GE.0)GO TO 43
  37:C
  38:C      CONSIDERAR UN NUMERO NEGATIVO
  39:C
  40:       NOFFS=65536+NOFFS
  41:43     CALL DTH(NOFFS,L4MC)
  42:       CALL NTAAC(L4MC,1,4)
  43:       CALL MOVER(L4MC,3,4,L80,20)
  44:49     CALL DTH(NQ4,L4MC)
  45:       CALL NTAAC(L4MC,1,4)
  46:       CALL MOVER(L4MC,1,4,L80,12)
  47:       WRITE(4'IT)L80
  48:       NQ4=NQ4+2
  49:       RETURN
  50:       END
  51:F
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : DTH
 5:C
 6:C      SUBRUTINA QUE CONVIERTE UN NUMERO DECIMAL A UN NUMERO
 7:C      HEXADECIMAL ALMACENADO EN UN ARREGLO DE CUATRO ELEMENTOS.
 8:C
 9:       SUBROUTINE DTH(NUM,NH)
10:       DIMENSION NH(4)
11:       A=NUM
12:       DO 30 I=1,3
13:       J=5-I
14:       A=A/16.0
15:       NA=A
16:       NH(J)=(A-NA)*16
17:30     A=NA
18:       NH(1)=A
19:       CALL NTAAC(NH,1,4)
20:       RETURN
21:       END
> LAST LINE
>
```

```
    1:*EXTENDED PRECISION
    2:*TWO WORD INTEGERS
    3:C
    4:C      SUBRUTINA : NTAAC
    5:C
    6:C      SUBRUTINA QUE CONVIERTE LOS ELEMENTOS DE UN ARREGLO, DE LA MODA -
    7:C      LIDAD DE ALMACENAMIENTO DE NUMERICO A ALFANUMERICO.
    8:C
    9:       SUBROUTINE NTAAC(NHNE,KKI,KKF)
   10:       IMPLICIT INTEGER*4 (H)
   11:       INTEGER NHNE(100),N1,N2,N3,N4,N5,N6,N7,N8,N9,N0,
   12:      6 A,B,C,D,E,F
   13:       DATA N1,N2,N3,N4,N5,N6,N7,N8/'1','2','3','4','5','6','7','8'/
   14:       DATA N9,N0,A,B,C,D,E,F/'9','0','A','B','C','D','E','F'/
   15:       DO 31 I=KKI,KKF
   16:       IF(NHNE(I).EQ.1)NHNE(I)=N1
   17:       IF(NHNE(I).EQ.2)NHNE(I)=N2
   18:       IF(NHNE(I).EQ.3)NHNE(I)=N3
   19:       IF(NHNE(I).EQ.4)NHNE(I)=N4
   20:       IF(NHNE(I).EQ.5)NHNE(I)=N5
   21:       IF(NHNE(I).EQ.6)NHNE(I)=N6
   22:       IF(NHNE(I).EQ.7)NHNE(I)=N7
   23:       IF(NHNE(I).EQ.8)NHNE(I)=N8
   24:       IF(NHNE(I).EQ.9)NHNE(I)=N9
   25:       IF(NHNE(I).EQ.0)NHNE(I)=N0
   26:       IF(NHNE(I).EQ.10)NHNE(I)=A
   27:       IF(NHNE(I).EQ.11)NHNE(I)=B
   28:       IF(NHNE(I).EQ.12)NHNE(I)=C
   29:       IF(NHNE(I).EQ.13)NHNE(I)=D
   30:       IF(NHNE(I).EQ.14)NHNE(I)=E
   31:       IF(NHNE(I).EQ.15)NHNE(I)=F
   32:31     CONTINUE
   33:       RETURN
   34:       END
--> LAST LINE
:=>
```

```
*EXTENDED PRECISION
*TWO WORD INTEGERS
C
C       SUBRUTINA : ATNAC
C
C       SUBRUTINA QUE PERMITE CONVERTIR LOS ELEMENTOS DE UN ARREGLO DE
C       LA MODALIDAD DE ALMACENAMIENTO ALFANUMERICO A LA DE NUMERICO.
C
        SUBROUTINE ATNAC(NHN,KI,KF)
        IMPLICIT INTEGER*4 (H)
        INTEGER NHN(100),N1,N2,N3,N4,N5,N6,N7,N8,N9,N0,
      6 A,B,C,D,E,F
        DATA N1,N2,N3,N4,N5,N6,N7,N8/'1','2','3','4','5','6','7','8'/
        DATA N9,N0,A,B,C,D,E,F/'9','0','A','B','C','D','E','F'/
        DO 30 I=KI,KF
        IF(NHN(I).EQ.N1)NHN(I)=1
        IF(NHN(I).EQ.N2)NHN(I)=2
        IF(NHN(I).EQ.N3)NHN(I)=3
        IF(NHN(I).EQ.N4)NHN(I)=4
        IF(NHN(I).EQ.N5)NHN(I)=5
        IF(NHN(I).EQ.N6)NHN(I)=6
        IF(NHN(I).EQ.N7)NHN(I)=7
        IF(NHN(I).EQ.N8)NHN(I)=8
        IF(NHN(I).EQ.N9)NHN(I)=9
        IF(NHN(I).EQ.N0)NHN(I)=0
        IF(NHN(I).EQ.A)NHN(I)=10
        IF(NHN(I).EQ.B)NHN(I)=11
        IF(NHN(I).EQ.C)NHN(I)=12
        IF(NHN(I).EQ.D)NHN(I)=13
        IF(NHN(I).EQ.E)NHN(I)=14
        IF(NHN(I).EQ.F)NHN(I)=15
30      CONTINUE
        RETURN
        END
-> LAST LINE
=>
```

```
 1:*EXTENDED PRECISION
 2:*TWO WORD INTEGERS
 3:C
 4:C      SUBRUTINA : SCHLB
 5:C
 6:C      SUBRUTINA QUE SE ENCARGA DE ENCONTRAR SI EXISTE UNA ETIQUETA
 7:C      O SIMBOLO EN LA TABLA DE SIMBOLOS, LAS VECES QUE ESTA ESTA
 8:C      DEFINIDA Y LA POSICION DE MEMORIA ASOCIADA A ELLA.
 9:C
10:       SUBROUTINE SCHLB(LB5,NLB,NB)
11:       DIMENSION LB5(5),LC5(5)
12:       NB=0
13:       NLB=0
14:       READ(2'1)NLR
15:       DO 30 I=2,NLR
16:       READ(2'I)LC5,NLC
17:       IF(LB5(1).NE.LC5(1).OR.LB5(2).NE.LC5(2).OR.
18:      & LB5(3).NE.LC5(3).OR.LB5(4).NE.LC5(4).OR.
19:      & LB5(5).NE.LC5(5))GO TO 30
20:       NLB=NLC
21:       NB=NB+1
22:30     CONTINUE
23:       RETURN
24:       END
-> LAST LINE
=>
```

```
   1:*EXTENDED PRECISION
   2:*TWO WORD INTEGERS
   3:C
   4:C      SUBRUTINA : HRNTF
   5:C
   6:C      SUBRUTINA QUE SE ENCARGA DE SEPARAR UN NUMERO ENTERO EN SUS
   7:C      DIGITOS CONSTITUTIVOS Y COLOCARLOS UNO POR UNO EN LOS ELE -
   8:C      MENTOS DE UN ARREGLO.
   9:C
  10:       SUBROUTINE HRNTF(NN,L,NI)
  11:       DIMENSION L(1)
  12:       N1=NN/10000.0
  13:       N2=(NN-N1*10000.0)/100.0
  14:       N3=NN-N1*10000.0-N2*100.0
  15:       NX=NI
  16:       L(NX)=N1
  17:       NX=NX+1
  18:       L(NX)=N2/10
  19:       NXM1=NX
  20:       NX=NX+1
  21:       L(NX)=N2-L(NXM1)*10
  22:       NX=NX+1
  23:       L(NX)=N3/10
  24:       NXM1=NX
  25:       NX=NX+1
  26:       L(NX)=N3-L(NXM1)*10
  27:       RETURN
  28:       END
-> LAST LINE
=>
```

```
    1:*EXTENDED PRECISION
    2:*TWO WORD INTEGERS
    3:C
    4:C      SUBRUTINA : DIR
    5:C
    6:C      SUBRUTINA QUE RELACIONA EL CODIGO DADO DE UNA PSEUDO OPERACION
    7:C      CON UN CODIGO NUMERICO PARA USO POSTERIOR DEL ENSAMBLADOR.
    8:C
    9:       SUBROUTINE DIR(L,NCD)
   10:       INTEGER ORG(3),EQU(3),RMB(3),NAM(3),END(3),FCC(3)
   11:       INTEGER FCB(3),L(3)
   12:       DATA ORG/'O','R','G'/,EQU/'E','Q','U'/,RMB/'R','M','B'/
   13:       DATA NAM/'N','A','M'/,END/'E','N','D'/,FCC/'F','C','C'/
   14:       DATA FCB/'F','C','B'/
   15:       NCD=0
   16:       CALL CMP(ORG,L,NM)
   17:       IF(NM.EQ.1)NCD=1
   18:       CALL CMP(EQU,L,NM)
   19:       IF(NM.EQ.1)NCD=2
   20:       CALL CMP(RMB,L,NM)
   21:       IF(NM.EQ.1)NCD=3
   22:       CALL CMP(NAM,L,NM)
   23:       IF(NM.EQ.1)NCD=4
   24:       CALL CMP(END,L,NM)
   25:       IF(NM.EQ.1)NCD=5
   26:       CALL CMP(FCC,L,NM)
   27:       IF(NM.EQ.1)NCD=6
   28:       CALL CMP(FCB,L,NM)
   29:       IF(NM.EQ.1)NCD=7
   30:       RETURN
   31:       END
--> LAST LINE
E=>
```

```
C-ERRS...STNO.C..... F O R T R A N   S O U R C E   S T A T E M E N T S  ........  IDENTFCN     **COMPILER MESSAGES**

      C
      C     PROGRAMA : A6800
      C
      C     PROGRAMA PRINCIPAL PARA EL ENSAMBLADOR DE PROGRAMAS ESCRITOS
      C     EN LENGUAJE ASSEMBLER DE LA MOTOROLA M6800.
      C
      C
      C     DEFINIR LOS ARCHIVOS A UTILIZARSE, ADEMAS DEFINIR ARREGLOS E
      C     INICIALIZAR VARIABLES.
      C
0001          DEFINE FILE 1(400,32,U,J1),2(500,12,U,J2)
0002          DEFINE FILE 3(250,6,U,J3),4(1000,160,U,J4)
0003          INTEGER*4 ORG(3),LCMP(3)
```

```
C-ERRS...STNO.C..... F O R T R A N  S O U R C E  S T A T E M E N T S ........ IDENTFCN    **COMPILER MESSAGES**
0004          DIMENSION L80(80),LD5(5),IH4H(4)
0005          DIMENSION LTS(17),L18(18),LCOD(16),LC6(6)
0006          DIMENSION LNET(5),LBL5(5),LB18(18)
0007          DATA LTS(1)-(10)/'T','A','B','L','A',' ','D','E',' ','S'/
0008          DATA LTS(11)-(17)/'I','M','B','O','L','O','S'/
0009          DATA NAST,IBLK/'*',' '/,ORG/'O','R','G'/
              C
              C     SECUENCIALIZAR LAS INSTRUCCIONES DEL PROGRAMA,
              C     Y BORRAR LAS AREAS DEL PROGRAMA OBJETO.
              C
0010          CALL CGPR
              C
              C     INICIALIZAR EL ARCHIVO DE ETIQUETAS, Y LEER EL NUMERO DE INSTR.
              C     DEL PROGRAMA MAS UNO
              C
0011          NREC=1
0012          WRITE(2'1)NREC
0013          READ(4'1)IP
              C
              C     CONSIDERAR LOS "DIRECTIVES" : NAM Y ORG EN ESE ORDEN.
              C
0014          READ(4'2)L80
0015          CALL VDF(L80,NTY)
0016          IF(NTY.EQ.4)GO TO 4080
0017          IT=2
0018          CALL PERR(L80,5,IT)
0019    4080  DO 300 NSH=1,2
0020          READ(4'3)L80
0021          CALL VDF(L80,NTY)
0022          IF(NTY.EQ.1)GO TO 4081
0023          IT=3
0024          CALL PERR(L80,6,IT)
0025          NQ4=0
0026          GO TO 4085
0027    4081  CALL SRORG(L80,NQ4,NER,3)
0028    4085  IPM1=IP-1
0029          IF(NQ4.LE.65535)GO TO 4086
0030          CALL PERR(L80,13,3)
0031    4086  CONTINUE
              C
              C     ANALIZAR EL RESTO DEL PROGRAMA ( A PARTIR DE LA LINEA NUMERO
              C     TRES HASTA LA LINEA DEL PROGRAMA QUE ESTA ANTES DEL "END" ).
              C
0032          DO 300 I=4,IPM1
0033          IT=I
0034          IF(NQ4.LE.65535)GO TO 4087
0035          CALL PERR(L80,13,IT)
0036    4087  CONTINUE
              C
              C     CONSIDERAR EL BLOQUE DE "DIRECTIVES" :
              C     ORG, EQU, RMB, FCB, Y FCC.
              C
0037          READ(4'I)L80
0038          IF(L80(25).EQ.NAST)GO TO 300
0039          CALL VDF(L80,NTY)
```

```
C-ERRS...STNO.C..... F O R T R A N  S O U R C E  S T A T E M E N T S ........ IDENTFCN    **COMPILER MESSAGES**

0040            IF(NTY.EQ.0)GO TO 400
0041            IF(NTY.EQ.2)GO TO 410
0042            IF(NTY.EQ.3)GO TO 420
0043            IF(NTY.EQ.5)GO TO 999
0044            IF(NTY.EQ.6)GO TO 4040
0045            IF(NTY.EQ.7)GO TO 4030
                C
                C     "PAUSE" SI NO ES NINGUNO DE LOS "DIRECTIVES" PREVISTOS.
                C
0046            IF(NTY.NE.1)PAUSE
                C
                C     NTY=1, "DIRECTIVE" ORG.
                C
0047            CALL SRORG(L80,NQ4,NER,IT)
0048            GO TO 300
0049      410   CALL SREQU(L80,NQ4,NER,IT,NSH)
0050            GO TO 300
0051      420   CALL SRRMB(L80,NQ4,NER,IT,NSH)
0052            GO TO 300
0053      4030  CALL SRFCB(L80,NQ4,NER,IT,NSH)
0054            GO TO 300
0055      4040  CALL SRFCC(L80,NQ4,NER,IT,NSH)
0056            GO TO 300
                C
                C     ENSAMBLAR LAS INSTRUCCIONES DEL PROGRAMA.
                C
0057      400   CONTINUE
0058            IF(L80(35).NE.IBLK)GO TO 482
0059            CALL MOVER(L80,32,34,LCMP,1)
0060            CALL CMP(LCMP,ORG,NMORG)
0061            IF(NMORG.EQ.0)GO TO 482
0062            CALL PERR(L80,1,IT)
0063            GO TO 300
0064      482   CONTINUE
                C
                C     VER SI EL CODIGO DE OPERACION ES VALIDO.
                C
0065            CALL VNMOP(L80,LCOD,NER)
0066            IF(NER.NE.0)GO TO 4001
0067            CALL PERR(L80,12,IT)
0068            GO TO 300
                C
                C     AVERIGUAR TIPO DE DIRECCIONAMIENTO.
                C
0069      4001  CALL MOVER(L80,38,55,L18,1)
0070            CALL INMED(L18,NX)
0071            IF(NX.EQ.1)GO TO 4002
0072            CALL NTYPE(L18,NC,NDEC,NCM,NPP)
0073            CALL LABEL(L18,NE,NDEC,LNET)
                C
                C     LLENAR LA TABLA DE SIMBOLOS
                C
0074            CALL MOVER(L80,25,42,LB18,1)
0075            CALL LABEL(LB18,NBE18,NBDEC,LBL5)
0076            IF(NBE18.EQ.0)GO TO 4004
```

```
C-ERRS...STNO.C..... FORTRAN SOURCE STATEMENTS ........ IDENTFCN    **COMPILER MESSAGES**
      0077           IF(NBE18,EQ,1)GO TO 4003
      0078           CALL PERR(L80,2,IT)
      0079           GO TO 4004
      0080    4003   IF(NSH.EQ.2)GO TO 4004
      0081           READ(2'1)NRLB
      0082           NRLB=NRLB+1
      0083           IF(NRLB,LE,520)GO TO 4005
      0084           CALL PERR(L80,13,IT)
      0085           GO TO 4004
      0086    4005   WRITE(2'NRLB)L8L5,NQ4
      0087           WRITE(2'1)NRLB
      0088    4004   CONTINUE
             C
      0089           CALL RELAT(L80,LCOD,NE,NX)
      0090           IF(NX.EQ.4)GO TO 4002
      0091           IF(NX.EQ.6)GO TO 4002
      0092           CALL INDEX(L18,NC,NDEQ,NCM,NPP,NX)
      0093           IF(NX.EQ.3)GO TO 4002
      0094           CALL DIREC(NDEQ,NE,NC,NX)
      0095           IF(NX.EQ.2.OR.NX.EQ.4)GO TO 4002
      0096           CALL IMPLI(L18,NC,LCOD,NX)
      0097           IF(NX.EQ.5)GO TO 4002
      0098           CALL PERR(L80,10,IT)
      0099           GO TO 300
      0100    4002   CONTINUE
      0101           CALL LCAR(LC6,LCOD,NX)
      0102           L80(17)=LC6(1)
      0103           L80(18)=LC6(2)
      0104           WRITE(4'IT)L80
             C
             C       COMPLETAR CODIGOS DE OPERANDO (PRIMERA CORRIDA)
             C
      0105           GO TO (4021,4022,4023,4022,4025,4026),NX
      0106    4021   CONTINUE
      0107           CALL COD1(L80,NQ4,IT)
      0108           GO TO 300
      0109    4022   CONTINUE
      0110           CALL COD24(L80,NQ4,LCOD,NSH,IT)
      0111           GO TO 300
      0112    4023   CONTINUE
      0113           CALL COD3(L80,NQ4,NSH,IT)
      0114           GO TO 300
      0115    4025   CONTINUE
      0116           CALL COD5(L80,NQ4,NSH,IT)
      0117           GO TO 300
      0118    4026   CONTINUE
      0119           CALL COD6(L80,NQ4,NSH,IT)
      0120    300    CONTINUE
      0121           READ(4'IP)L80
      0122           CALL VOF(L80,NTY)
      0123           IF(NTY,EQ,5)GO TO 999
      0124           CALL PERR(L80,1,IP)
             C
             C       TITULARES DE LA SALIDA DEL CROSS-ASSEMBLER
             C
```

```
C-ERRS...STNO.C..... F O R T R A N  S O U R C E  S T A T E M E N T S ........ IDENTFCN    **COMPILER MESSAGES**

0125      999   WRITE(1,257)
0126      257   FORMAT(/,5X,'INGRESE UN ; 1 PARA SALIDA EN PANTALLA',//,
                6 18X,'5 PARA SALIDA EN IMPRESORA',//,'9')
0127            READ(6,10,END=88)NWP
0128      10    FORMAT(I1)
0129            WRITE(NWP,255)
0130      255   FORMAT(1H1,/,5X,'ESCUELA POLITECNICA NACIONAL',/,
                6 5X,'FACULTAD DE INGENIERIA ELECTRICA',//,
                6 5X,'TESIS DE GRADO : 25 DE JUNIO DE 1982',///,
                6 5X,'REALIZADO POR :-HIRIAM HERNANDEZ ALVAREZ',/,
                6 5X,'DIRECTOR DE TESIS : ING. EDGAR P. TORRES P.',/)
0131            WRITE(NWP,219)
0132      219   FORMAT(//,5X,'SALIDA DEL CROSS-ASSEMBLER DE LA MOTOROLA',
                6 ' M6800',/,5X,47('='),//)
          C     CREACION DE LA TABLA DE SIMBOLOS EN EL ARCHIVO FUENTE
0133            DO 310 I=2,IP
0134            READ(4'I)L80
0135            WRITE(NWP,260)L80
0136      260   FORMAT(80A1)
0137      310   CONTINUE
0138            WRITE(NWP,261)
0139      261   FORMAT(/,5X,'TABLA DE SIMBOLOS',//)
0140            READ(2'1)NREC
0141            DO 320 I=2,NREC
0142            READ(2'I)LD5,IHD5
0143            CALL DTH(IHD5,IH4H)
0144            CALL NTAAC(IH4H,1,4)
0145            WRITE(NWP,262)LD5,IH4H
0146      262   FORMAT(5X,5A1,3X,4A1)
0147      320   CONTINUE
0148            NERCT=0
0149            DO 330 I=2,IP
0150            READ(4'I)L80
0151            IF(L80(1).EQ.NAST)NERCT=NERCT+1
0152      330   CONTINUE
0153            IF(NERCT.EQ.0)GO TO 4082
0154            WRITE(NWP,22)NERCT
0155      22    FORMAT(/,5X,'*** ',I4,' ERRORES DETECTADOS EN EL ENSAMBLAJE',
                6 ' DE ESTE PROGRAMA  ***'/)
0156            GO TO 4083
0157      4082  WRITE(NWP,233)
0158      233   FORMAT(//,5X,'NINGUN ERROR DETECTADO EN EL ENSAMBLAJE DE',
                6' ESTE PROGRAMA',///,5X,'FIN DE UN ENSAMBLAJE EXITOSO',//)
0159            GO TO 88
0160      4083  WRITE(NWP,221)
0161      221   FORMAT(/,5X,'FIN DE LA SALIDA DEL CROSS-ASSEMBLER',/)
0162      88    CALL EXIT
0163            END

VARIABLE-ALLOCATIONS
 EQUIVALENCES & INTERNAL VARIABLES
    J1(I*4D )=001C      J2(I*4D )=001E      J3(I*4D )=0020      J4(I*4D )=0022      NREC(I*4D )=0024
    IP(I*4D )=0026      NTY(I*4D )=0028     IT(I*4D )=002A      NSH(I*4D )=002C     NQ4(I*4D )=002E
   NER(I*4D )=0030     IPM1(I*4D )=0032      I(I*4D )=0034     NAST(I*4D )=0036      IBLK(I*4D )=0038
 NHORG(I*4D )=003A       NX(I*4D )=003C     NC(I*4D )=003E     NDEQ(I*4D )=0040      NCM(I*4D )=0042
   NPP(I*4D )=0044       NE(I*4D )=0046   NOEC(I*4D )=0048   NBE18(I*4D )=004A   NBDEC(I*4D )=004C
```

```
NRLB(I*4D )=004E    NWP(I*4D )=0050    IHD5(I*4D )=0052    NERCT(I*4D )=0054    L80(I*4D )=00FC-005E
LD5(I*4D )=0106-00FE    IH4H(I*4D )=010E-0108    LTS(I*4D )=0130-0110    L18(I*4D )=0154-0132    LCOD(I*4D )=0174-0156
LC6(I*4D )=0180-0176    LNET(I*4D )=018A-0182    LBL5(I*4D )=0194-018C    LB18(I*4D )=01B8-0196    ORG(I*4D )=01BE-01BA
LCMP(I*4D )=01C4-01C0

STATEMENT ALLOCATIONS
  257=01ED    10=0219    255=021B    219=028C    260=0282    261=0285    262=02C3    22=02CA    233=02F2    221=0327
 4080=038D   4081=0384   4085=038A   4086=03CD   4087=03E2    410=0438    420=0441   4030=044A   4040=0453    400=045C
  482=047F   4001=0493   4003=04D7   4005=04FB   4004=050F.  4002=0562   4021=0587   4022=058E   4023=0597   4025=059F
 4026=05A7    300=05AD    999=0504    310=0602    320=063A    330=065F   4082=0678   4083=067F     88=0684

FEATURES SUPPORTED
 TWO WORD INTEGERS
 EXTENDED PRECISION
 IOCS-
   DISK
   KEYBOARD
   1403 PRINTER
   TYPEWRITER
   CARD

CALLED SUBPROGRAMS
 CGPR    VDF     PERR    SRORG   SREQU   SRRMB   SRFCB   SRFCC   MOVER   CMP     VNMOP   INMED   NTYPE   LABEL   RELAT
 INDEX   DIREC   IMPLI   LCAR    COD1    COD24   COD3    COD5    COD6    DTH     NTAAC   INITS   ELD     ESTO    TYPEZ
 CARDZ   SRED    SWRT    SCOMP   SFIO    SIOAI   SIOI    PRNZ    PAUSE   SDFIO   SDRED   SDWRT   SDCOM   SEOF    SDAI
 SDI     ISUB    LRLE    LRNE    LREQ    DQTST   I4TST

INTEGER CONSTANTS
      1=01C6      2=01C8      4=01CA      5=01CC      3=01CE      6=01D0      0=01D2
  65535=01D4     13=01D6      7=01D8     32=01DA     34=01DC     12=01DE     38=01E0
     55=01E2     25=01E4     42=01E6     52=01E8     10=01EA

CORE REQUIREMENTS FOR - A6800
BLANK COMMON-    0,    VARIABLES AND TEMPORARIES-    454,    CONSTANTS AND PROGRAM-    1216

END OF SUCCESSFUL COMPILATION.                                  COPYRIGHT DNA SYSTEMS, INC.
```

```
 SYMBOL   USAGE        MODE LNGTH   REFERENCES

 CGPR     SUBROUTINE              0010,1
 CMP      SUBROUTINE              0060,1
 COD1     SUBROUTINE              0107,1
 COD24    SUBROUTINE              0110,1
 COD3     SUBROUTINE              0113,1
 COD5     SUBROUTINE              0116,1
 COD6     SUBROUTINE              0119,1
 DIREC    SUBROUTINE              0094,1
 DTH      SUBROUTINE              0143,1
 EXIT     SUBROUTINE              0162,1
 I        SCALAR      INTEGER*4   0032,1  0033,1  0037,1  0133,1  0134,1  0141,1  0142,1  0149,1  0150,1
 IBLK     SCALAR      INTEGER*4   0009,1  0058,1
 IHD5     SCALAR      INTEGER*4   0142,1  0143,1
 IH4H     VECTOR      1*4(0004)   0004,1  0143,1  0144,1  0145,1
 IMPLI    SUBROUTINE              0096,1
 INDEX    SUBROUTINE              0092,1
 INMED    SUBROUTINE              0070,1
 IP       SCALAR      INTEGER*4   0013,1  0028,1  0121,1  0124,1  0133,1  0149,1
 IPH1     SCALAR      INTEGER*4  *0028,1  0032,1
 IT       SCALAR      INTEGER*4  *0017,1  0018,1 *0023,1  0024,1 *0033,1  0035,1  0047,1  0049,1  0051,1  0053,1  0055,1
                                  0062,1  0067,1  0078,1  0084,1  0098,1  0104,1  0107,1  0110,1  0113,1  0116,1  0119,1

 J1       SCALAR      INTEGER*4   0001,1
 J2       SCALAR      INTEGER*4   0001,1
 J3       SCALAR      INTEGER*4   0002,1
 J4       SCALAR      INTEGER*4   0002,1
 LABEL    SUBROUTINE              0073,1  0075,1
 LBL5     VECTOR      I*4(0005)   0006,1  0075,1  0086,1
 LB18     VECTOR      I*4(0018)   0006,1  0074,1  0075,1
 LCAR     SUBROUTINE              0101,1
 LCMP     VECTOR      I*4(0003)   0003,1  0059,1  0060,1
 LCOD     VECTOR      I*4(0016)   0005,1  0065,1  0089,1  0096,1  0101,1  0110,1
 LC6      VECTOR      I*4(0006)   0005,1  0101,1  0102,1  0103,1
 LD5      VECTOR      I*4(0005)   0004,1  0142,1  0145,1
 LNET     VECTOR      I*4(0005)   0006,1  0073,1
 LTS      VECTOR      I*4(0017)   0005,1  0007,1  0008,1
 L18      VECTOR      I*4(0018)   0005,1  0069,1  0070,1  0072,1  0073,1  0092,1  0096,1
 L80      VECTOR      I*4(0080)   0004,1  0014,1  0015,1  0018,1  0020,1  0021,1  0024,1  0027,1  0030,1  0035,1  0037,1
                                  0038,1  0039,1  0047,1  0049,1  0051,1  0053,1  0055,1  0058,1  0059,1  0062,1  0065,1
                                  0067,1  0069,1  0074,1  0078,1  0084,1  0089,1  0098,1 *0102,1 *0103,1  0104,1  0107,1
                                  0110,1  0113,1  0116,1  0119,1  0121,1  0122,1  0124,1  0134,1  0135,1  0150,1  0151,1
 MOVER    SUBROUTINE              0059,1  0069,1  0074,1
 NAST     SCALAR      INTEGER*4   0009,1  0038,1  0151,1
 NBDEC    SCALAR      INTEGER*4   0075,1
 NBEIB    SCALAR      INTEGER*4   0075,1  0076,1  0077,1
 NC       SCALAR      INTEGER*4   0072,1  0092,1  0094,1  0096,1
 NCM      SCALAR      INTEGER*4   0072,1  0092,1
 NDEC     SCALAR      INTEGER*4   0073,1
 NDEQ     SCALAR      INTEGER*4   0072,1  0092,1  0094,1
 NE       SCALAR      INTEGER*4   0073,1  0089,1  0094,1
 NER      SCALAR      INTEGER*4   0027,1  0047,1  0049,1  0051,1  0053,1  0055,1  0065,1  0066,1
 NERCT    SCALAR      INTEGER*4  *0148,1 *0151,2  0153,1  0154,1
 NMORG    SCALAR      INTEGER*4   0060,1  0061,1
 NPP      SCALAR      INTEGER*4   0072,1  0092,1
 NQ4      SCALAR      INTEGER*4  *0025,1  0027,1  0029,1  0034,1  0047,1  0049,1  0051,1  0053,1  0055,1  0086,1  0107,1
                                  0110,1  0113,1  0116,1  0119,1
 NREC     SCALAR      INTEGER*4  *0011,1  0012,1  0140,1  0141,1
```

SYMBOL	USAGE	MODE LNGTH	REFERENCES
NRLB	SCALAR	INTEGER*4	0081,1 *0082,2 0083,1 0086,1 0087,1
NSH	SCALAR	INTEGER*4	0019,1 0049,1 0051,1 0053,1 0055,1 0080,1 0110,1 0113,1 0116,1 0119,1
NTAAC	SUBROUTINE		0144,1
NTY	SCALAR	INTEGER*4	0015,1 0016,1 0021,1 0022,1 0039,1 0040,1 0041,1 0042,1 0043,1 0044,1 0045,1 0046,1 0122,1 0123,1
NTYPE	SUBROUTINE		0072,1
NWP	SCALAR	INTEGER*4	0127,1 0129,1 0131,1 0135,1 0138,1 0145,1 0154,1 0157,1 0160,1
NX	SCALAR	INTEGER*4	0070,1 0071,1 0089,1 0090,1 0091,1 0092,1 0093,1 0094,1 0095,2 0096,1 0097,1 0101,1 0105,1
ORG	VECTOR	I*4(0003)	0003,1 0009,1 0060,1
PERR	SUBROUTINE		0018,1 0024,1 0030,1 0035,1 0062,1 0067,1 0078,1 0084,1 0098,1 0124,1
RELAT	SUBROUTINE		0089,1
SREQU	SUBROUTINE		0049,1
SRFCB	SUBROUTINE		0053,1
SRFCC	SUBROUTINE		0055,1
SRORG	SUBROUTINE		0027,1 0047,1
SRRMB	SUBROUTINE		0051,1
U	SCALAR	REAL*6	0001,2 0002,2
VDF	SUBROUTINE		0015,1 0021,1 0039,1 0122,1
VNMOP	SUBROUTINE		0065,1
10	FORMAT NO.		0127,1 *0128,1
22	FORMAT NO.		0154,1 *0155,1
88	STATMT NO.		0127,1 0159,1 *0162,1
219	FORMAT NO.		0131,1 *0132,1
221	FORMAT NO.		0160,1 *0161,1
233	FORMAT NO.		0157,1 *0158,1
255	FORMAT NO.		0129,1 *0130,1
257	FORMAT NO.		0125,1 *0126,1
260	FORMAT NO.		0135,1 *0136,1
261	FORMAT NO.		0138,1 *0139,1
262	FORMAT NO.		0145,1 *0146,1
300	STATMT NO.		0019,1 0032,1 0038,1 0048,1 0050,1 0052,1 0054,1 0056,1 0063,1 0068,1 0099,1 0108,1 0111,1 0114,1 0117,1 *0120,1
310	STATMT NO.		0133,1 *0137,1
320	STATMT NO.		0141,1 *0147,1
330	STATMT NO.		0149,1 *0152,1
400	STATMT NO.		0040,1 *0057,1
410	STATMT NO.		0041,1 *0049,1
420	STATMT NO.		0042,1 *0051,1
482	STATMT NO.		0058,1 0061,1 *0064,1
999	STATMT NO.		0043,1 0123,1 *0125,1
4001	STATMT NO.		0066,1 *0069,1
4002	STATMT NO.		0071,1 0090,1 0091,1 0093,1 0095,1 0097,1 *0100,1
4003	STATMT NO.		0077,1 *0080,1
4004	STATMT NO.		0076,1 0079,1 0080,1 0085,1 *0088,1
4005	STATMT NO.		0083,1 *0086,1
4021	STATMT NO.		0105,1 *0106,1
4022	STATMT NO.		0105,2 *0109,1
4023	STATMT NO.		0105,1 *0112,1
4025	STATMT NO.		0105,1 *0115,1
4026	STATMT NO.		0105,1 *0118,1
4030	STATMT NO.		0045,1 *0053,1
4040	STATMT NO.		0044,1 *0055,1
4080	STATMT NO.		0016,1 *0019,1
4081	STATMT NO.		0022,1 *0027,1
4082	STATMT NO.		0153,1 *0157,1

```
SYMBOL   USAGE    MODE LNGTH   REFERENCES

4083     STATMT NO.           0156,1 *0160,1
4085     STATMT NO.           0026,1 *0028,1
4086     STATMT NO.           0029,1 *0031,1
4087     STATMT NO.           0034,1 *0036,1

         0108 SYMBOL TABLE ENTRIES
```

CONSTANTS	MODE LNGTH	REFERENCES
0	INTEGER*4	0025,1 0040,1 0061,1 0066,1 0076,1 0148,1 0153,1
1	INTEGER*4	0001,1 0007,1 0011,1 0012,1 0013,1 0019,1 0022,1 0028,1 0046,1 0059,1 0062,1 0069,1 0071,1 0074,1 0077,1 0081,1 0082,1 0087,1 0102,1 0124,1 0125,1 0140,1 0144,1 0151,2
2	INTEGER*4	0001,1 0012,1 0014,1 0017,1 0019,1 0041,1 0078,1 0080,1 0081,1 0086,1 0087,1 0095,1 0103,1 0133,1 0140,1 0141,1 0142,1 0149,1
3	INTEGER*4	0002,1 0003,2 0020,1 0023,1 0027,1 0030,1 0042,1 0093,1
4	INTEGER*4	0002,1 0003,1 0004,1 0013,1 0014,1 0016,1 0020,1 0032,1 0037,1 0090,1 0095,1 0104,1 0121,1 0134,1 0144,1 0150,1
5	INTEGER*4	0004,1 0006,2 0018,1 0043,1 0097,1 0123,1
6	INTEGER*4	0002,1 0005,1 0024,1 0044,1 0091,1 0127,1
7	INTEGER*4	0045,1
10	INTEGER*4	0007,1 0098,1
11	INTEGER*4	0008,1
12	INTEGER*4	0001,1 0067,1
13	INTEGER*4	0030,1 0035,1 0084,1
16	INTEGER*4	0005,1
17	INTEGER*4	0005,1 0008,1 0102,1
18	INTEGER*4	0005,1 0006,1 0103,1
25	INTEGER*4	0038,1 0074,1
32	INTEGER*4	0001,1 0059,1
34	INTEGER*4	0059,1
35	INTEGER*4	0058,1
38	INTEGER*4	0069,1
42	INTEGER*4	0074,1
55	INTEGER*4	0069,1
80	INTEGER*4	0004,1
160	INTEGER*4	0002,1
250	INTEGER*4	0002,1
400	INTEGER*4	0001,1
500	INTEGER*4	0001,1
520	INTEGER*4	0083,1
1000	INTEGER*4	0002,1
65535	INTEGER*4	0029,1 0034,1

A P E N D I C E C.

C.1 FORMA GENERAL DE UN PROGRAMA EN EL ASSEMBLER DE LA M6800.

Un programa en assembler es una secuencia de líneas llamadas instrucciones que siguen reglas especiales que hacen posible su programación y traducción. Estas reglas conforman la gramática del Assembler. Existen diversos niveles de Assembler y entre unos y otros las reglas que los manejan pueden variar poco o mucho y brindar más o menos facilidades al programador. En un caso extremo las reglas gramaticales pueden ser totalmente arbitrarias, siempre que coincidan en los códigos de máquina a ser generados, que deben ser compatibles con el procesador respectivo.

En este trabajo el assembler desarrollado tiene una gramática bastante próxima a lo que se entiende como assembler standard y brinda un buen nivel de ventaja al usuario.

Se verán a continuación las particularidades del assembler usado. Esto se inicia con una descripción

de formatos de instrucciones y el orden de ejecución de las mismas.

Se puede afirmar que, las instrucciones del assembler son ejecutadas en orden, a menos que existan saltos o transferencias de control

Cada instrucción está constituída por campos, como se muestra en la tabla C.1.

Fig. C.1 Una oración assembler.

Etiqueta	Código nmotécnico del operador	Campo de dirección u operando	Comentarios

Ejemplo C.1

Inicio * * *	ADDA	$60	Carge el primer número en la dirección hexadecimal 60.

El assembler utilizado tiene un formato fijo con ciertas columnas reservadas para un campo específico. Este tipo de disposición tiene la ventaja de no requerir símbolos especiales para separar campos.

El formato de entrada utilizado es el siguiente:

Fig. C.2 Formato de entrada

Etiqueta			Operador			Operando o dirección		Comentario
25	29	bb	32	35	bb	38	b	80

A continuación se van a estudiar por separado las partes que conforman una instrucción de este assembler.

C.1.1 Etiquetas.

El campo de etiquetas permite asignar un nombre o una dirección de una instrucción o dato. Luego se puede usar esta etiqueta para referirse a dicha dirección en otra instrucción.

La etiqueta puede tener de uno a cinco caracteres que pueden ser numéricos (0,1...,9), o letras (A,B,...,Z). El primer carácter debe ser una letra, a excepción de X que está reservada para el manejo del registro índice.

Una etiqueta determinada puede ser usada para referir-

nes o pseudo-operaciones.

Una pseudo-operación, es una directiva que aparece en el campo de los operadores que se diferencian del resto de códigos nmotécnicos en que no se traducen a lenguaje binario.

Las pseudo-operaciones asignan áreas en memoria, definen símbolos, designan áreas de RAM para almacenamiento temporal de datos y realizan otras funciones de este tipo. Generalmente se colocan todas las pseudo-operaciones agrupadas al inicio del programa.

En el assembler usado, las pseudo-operaciones permitidas son: NAME, EQUATE, ORIGIN, RESERVE, END, FORM-CONSTANT-BYTE, .FORM-CONSTANT-CHARACTER. Discutiremos brevemente su utilización, porque sus funciones normalmente resultan bastante obvias.

- NAME, sirve para nominar al programa, es necesariamente la primera instrucción del mismo. El nmotécnico usado es NAM y debe ir seguido de un nombre de programa.

- EQUATE, permite al programador igualar nombres con direcciones.
  El nmotécnico que corresponde es EQU. Los nombres

pueden referirse a direcciones, direcciones iniciales, direcciones fijas, etc. Esta pseudo-operación, asigna el valor numérico del campo del operando al nombre en el campo de etiqueta. Ilustremos su uso con un ejemplo que contiene dos instrucciones con el código EQU.

Ej. C.2

```
INICI    EQU    4
ETIQU    EQU    60
```

Se iguala los nombres INICI y ETIQU etiquetas de direcciones, a los valores 4 y 60 respectivamente. En el programa ensamblado a la izquierda de EQU, no aparece el contador de memoria, puesto que no gasta ninguna localidad de memoria.

- ORIGIN, permite ensamblar programas, subrutinas ó datos en cualquier lugar de memoria. El nmotécnico correspondiente es ORG. La segunda instrucción del programa debe ser necesariamente ORG, pero pueden haber más "ORG" en cualquier sitio a lo largo del programa.

  El assembler mantiene un contador de localidades que contiene la localidad de memoria de la siguiente instrucción o item de datos a ser procesados. Una pseudo-operación ORG, casu que se cambie el valor del contador de localidades. Se verá un ejemplo:

Ej. C.5

```
FORMA    FCC    ,
FORM1    FCB    5
```

Indica que se ingresa la representación en ASCII de una coma en la dirección etiquetada por FORMA y que se guarda en la dirección etiquetada por FORM1 la constante numérica 5.

- END: finaliza el programa en assembler. El nmotécnico correspondiente es END. Debe ser la última instrucción del programa.

Una pseudo-operación que no acepta etiquetas, es END, pues el significado de las mismas no sería claro.

Igualmente si un renglón que contiene una directiva ORG tiene etiqueta, ésta produce una condición de error.

Hay pseudo-operaciones como RMB, FCC, EQU, FCB, que siempre tienen etiquetas. EQU requiere obligatoriamente de una etiqueta, dado que su propósito es definir su significado. RMB y FCC las necesitan para direccionar los bytes de memoria con que están trabajando.

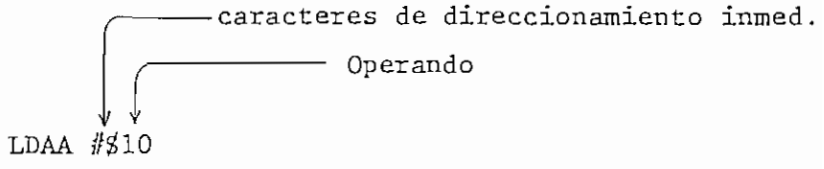
A continuación se va a estudiar cada uno de éstos, indicando su forma de uso y sus efectos:

C.2.1 Inmediato

El operando está contenido en el segundo byte de la instrucción, ó en el segundo y tercer byte de la instrucción. Con este direccionamiento se tienen, por lo tanto, instrucciones de dos o tres bytes.

Se reconoce encontrando el carácter "#" antes del operando. Este direccionamiento no acepta etiquetas ni acepta un nivel de suma (Operando no puede ser, por ejemplo: # LAB 1 ó # A1 + 2).

Ej. C.9



Indica que se "carge el acumulador A con el número hexadecimal 10."

Fig. C. 3 Bytes de la instrucción

Operador	1er. byte
Operando	2do. byte

o también:

Operador	1er. byte
8 bits más significativos operando	2do. byte
8 bits menos significativos operando	3er. byte.

C.2.2 Directo

La dirección del operando está contenido en el segundo byte de la instrucción. Permite al usuario direccionar desde la localidad de memoria cero hasta la 255. Las instrucciones son de 2 bytes. Se permiten también etiquetas para señalar las direcciones que de todos modos deben ser menores a 255. Para reconocer este tipo de direccionamiento, no se debe colocar ningún símbolo especial en el operando, sino directamente la dirección de memoria que se va a manejar.

Ej. C. 10

```
        LDAA    $C1     dirección del
                        operando
```

Significa "carge en el acumulador A el contenido de la localidad de memoria C1 (hexadecimal).

```
        ADDA    ETIQ
```

Pide al assembler que genere una instrucción ADD que sumará el contenido de la dirección ETIQ al contenido del acumulador A.

Fig. C.5 Bytes de la instrucción

Operador	1er. byte
8 bits más significativos dirección del operando	2do. byte
8 bits menos significativos dirección del operando	3er. byte.

C.2.4 Indexado.

La dirección contenida en el segundo byte de la instrucción, es sumado al registro índice. El resultado se usa para direccionar memoria. La dirección modificada se coloca en un registro temporal, de modo que el contenido del registro índice no cambia. Es una instrucción de 2 bytes.

El direccionamiento indexado se reconoce por una X en la primera posición del operando, aunque también puede tener la forma: expresión,X en el campo de operando de la instrucción. En el primer caso la dirección del operando viene dada por el contenido del registro índice' En el segundo caso la dirección del operando es la suma de la expresión más el contenido.

Ej. C.12

```
                    cantidad a ser sumada al contenido del
ADDA $30,X          registro índice.
```

Pide al assembler que genere una instrucción ADDA que sumará al acumulador A el valor presentado en la localidad de memoria especificada por la suma de 30 (hexadecimal) más el contenido del registro índice. Si el registro índice contiene 0316 (hexadecimal), el diagrama siguiente muestra el significado de la instrucción.

(A) = (A) + (X) + ($30)

(A) = (A) + ($0316+$30) = (A) + ($0346)

Fig. C.6 Bytes de la instrucción

	Operador	1er. byte
Expresión a ser sumado al registro índice		2do. byte

ó también:

	Operador	1er. byte
	ØØ	2do. byte

C.2.5 Implícito

En este direccionamiento la instrucción ya da la dirección del operando, por ejemplo: puntero de stack, re-

La mayoría de programas reales hacen una serie de tareas, muchas de las cuales son las mismas o pueden ser comunes a varios programas diferentes. Las subrutinas permiten formular estas tareas una sola vez y hacerlo de modo que puedan ser utilizadas en diferentes partes de un programa o en otros programas. La secuencia de instrucciones que conforman una subrutina es programada una sola vez, probada y luego usada repetidamente. Ciertas subrutinas especiales de uso bastante continuado, como las que realizan funciones trigonométricas, funciones de entrada/salida, entre otras, pueden formar parte de una librería de sistema.

La M6800 tiene instrucciones para transferir el control a subrutinas y restaurar el control al programa principal. Las instrucciones Jump-to-Subroutine(JSR) o Branch-to-Subroutine (BSR) rescatan el valor anterior del contador del programa en la memoria Random del Stack antes de ubicar en él, la dirección inicial de la subrutina. La instrucción Return-from-Subrutine (RTS) toma el valor anterior del contador del programa del Stack y lo pone de regreso en el contador del programa. El efecto que se tiene es de transferir el control del programa, primero a la subrutina y luego de regreso al programa principal. Una subrutina también puede transferir el control a otra sub-

```
WHAT FILE  3?>     ASCII
    2 :          20
    3 : !        21
    4 : "        22
    5 : #        23
    6 : $        24
    7 : %        25
    8 : &        26
    9 : '        27
   10 : (        28
   11 : )        29
   12 : *        2A
   13 : +        2B
   14 : ,        2C
   15 : -        2D
   16 : .        2E
   17 : /        2F
   18 : 0        30
   19 : 1        31
   20 : 2        32
   21 : 3        33
   22 : 4        34
   23 : 5        35
   24 : 6        36
   25 : 7        37
   26 : 8        38
   27 : 9        39
   28 : :        3A
   29 : ;        3B
   30 : <        3C
   31 : =        3D
   32 : >        3E
   33 : ?        3F
   34 : A        41
   35 : B        42
   36 : C        43
   37 : D        44
   38 : E        45
   39 : F        46
   40 : G        47
   41 : H        48
   42 : I        49
   43 : J        4A
   44 : K        4B
   45 : L        4C
   46 : M        4D
   47 : N        4E
   48 : O        4F
   49 : P        50
   50 : Q        51
   51 : R        52
   52 : S        53
   53 : T        54
   54 : U        55
   55 : V        56
   56 : W        57
   57 : X        58
   58 : Y        59
   59 : Z        5A
   60 : [        5B
```

```
TANEM

WHAT FILE  1?>      NMONI


      NOMENCLATURA :
      ==============

      NMON = CODIGO NMONICO
      D1 = INMEDIATO
      D2 = DIRECTO
      D3 = INDEXADO
      D4 = EXTENDIDO
      D5 = IMPLICITO
      D6 = RELATIVO


      INICIALIZACION DEL ARCHIVO ? (S/N)


      LECTURA DE ARCHIVO ? (S/N)
S


      107  REGISTROS CON INFORMACION VALIDA


         NMON D1  D2  D3  D4  D5  D6

        2 ADDA 8B  9B  AB  BB
        3 ADDB CB  DB  EB  FB
        4 ABA                   1B
        5 ADCA 89  99  A9  B9
        6 ADCB C9  D9  E9  F9
        7 ANDA 84  94  A4  B4
        8 ANDB C4  D4  E4  F4
        9 BITA 85  95  A5  B5
       10 BITB C5  D5  E5  F5
       11 CLR          6F  7F
       12 CLRA                  4F
       13 CLRB                  5F
       14 CMPA 81  91  A1  B1
       15 CMPB C1  D1  E1  F1
       16 CBA                   11
       17 COM          63  73
       18 COMA                  43
       19 COMB                  53
       20 NEG          60  70
       21 NEGA                  40
       22 NEGB                  50
       23 DAA                   19
       24 DEC          6A  7A
       25 DECA                  4A
       26 DECB                  5A
       27 EORA 88  98  A8  B8
       28 EORB C8  D8  E8  F8
       29 INC          6C  7C
       30 INCA                  4C
       31 INCB                  5C
       32 LDAA 86  96  A6  B6
```

33	LDAB	C6	D6	E6	F6		
34	ORAA	8A	9A	AA	BA		
35	ORAB	CA	DA	EA	FA		
36	PSHA					36	
37	PSHB					37	
38	PULA					32	
39	PULB					33	
40	ROL			69	79		
41	ROLA					49	
42	ROLB					59	
43	ROR			66	76		
44	RORA					46	
45	RORB					56	
46	ASL			68	78		
47	ASLA					48	
48	ASLB					58	
49	ASR			67	77		
50	ASRA					47	
51	ASRB					57	
52	LSR			64	74		
53	LSRA					44	
54	LSRB					54	
55	STAA		97	A7	B7		
56	STAB		D7	E7	F7		
57	SUBA	80	90	A0	B0		
58	SUBB	C0	D0	E0	F0		
59	SBA					10	
60	SBCA	82	92	A2	B2		
61	SBCB	C2	D2	E2	F2		
62	TAB					16	
63	TBA					17	
64	TST			6D	7D		
65	TSTA					4D	
66	TSTB					5D	
67	CPX	8C	9C	AC	BC		
68	DEX					09	
69	DES					34	
70	INX					08	
71	INS					31	
72	LDX	CE	DE	EE	FE		
73	LDS	8E	9E	AE	BE		
74	STX		DF	EF	FF		
75	STS		9F	AF	BF		
76	TXS					35	
77	TSX					30	
78	BRA						20
79	BCC						24
80	BCS						25
81	BEQ						27
82	BGE						2C
83	BGT						2E
84	BHI						22
85	BLE						2F
86	BLS						23
87	BLT						2D
88	BMI						2B
89	BNE						26
90	BVC						28
91	BVS						29
92	BPL						2A
93	BSR						8D
94	JMP			6E	7E		
95	JSR			AD	BD		
96	NOP					01	
97	RTI					3B	
98	RTS					39	

```
TANEM

WHAT FILE  1?>      NMONI


      NOMENCLATURA :
      ==============

      NMON = CODIGO NMONICO
      D1 = INMEDIATO
      D2 = DIRECTO
      D3 = INDEXADO
      D4 = EXTENDIDO
      D5 = IMPLICITO
      D6 = RELATIVO


      INICIALIZACION DEL ARCHIVO ? (S/N)


     LECTURA DE ARCHIVO ? (S/N)
S


      107  REGISTROS CON INFORMACION VALIDA


         NMON D1 D2 D3 D4 D5 D6

        2 ADDA 8B 9B AB BB
        3 ADDB CB DB EB FB
        4 ABA                  1B
        5 ADCA 89 99 A9 B9
        6 ADCB C9 D9 E9 F9
        7 ANDA 84 94 A4 B4
        8 ANDB C4 D4 E4 F4
        9 BITA 85 95 A5 B5
       10 BITB C5 D5 E5 F5
       11 CLR         6F 7F
       12 CLRA                 4F
       13 CLRB                 5F
       14 CMPA 81 91 A1 B1
       15 CMPB C1 D1 E1 F1
       16 CBA                  11
       17 COM         63 73
       18 COMA                 43
       19 COMB                 53
       20 NEG         60 70
       21 NEGA                 40
       22 NEGB                 50
       23 DAA                  19
       24 DEC         6A 7A
       25 DECA                 4A
       26 DECB                 5A
       27 EORA 88 98 A8 B8
       28 EORB C8 D8 E8 F8
       29 INC         6C 7C
       30 INCA                 4C
       31 INCB                 5C
       32 LDAA 86 96 A6 B6
```

33	LDAB	C6	D6	E6	F6			
34	ORAA	8A	9A	AA	BA			
35	ORAB	CA	DA	EA	FA			
36	PSHA					36		
37	PSHB					37		
38	PULA					32		
39	PULB					33		
40	ROL			69	79			
41	ROLA					49		
42	ROLB					59		
43	ROR			66	76			
44	RORA					46		
45	RORB					56		
46	ASL			68	78			
47	ASLA					48		
48	ASLB					58		
49	ASR			67	77			
50	ASRA					47		
51	ASRB					57		
52	LSR			64	74			
53	LSRA					44		
54	LSRB					54		
55	STAA		97	A7	B7			
56	STAB		D7	E7	F7			
57	SUBA	80	90	A0	B0			
58	SUBB	C0	D0	E0	F0			
59	SBA					10		
60	SBCA	82	92	A2	B2			
61	SBCB	C2	D2	E2	F2			
62	TAB					16		
63	TBA					17		
64	TST			6D	7D			
65	TSTA					4D		
66	TSTB					5D		
67	CPX	8C	9C	AC	BC			
68	DEX					09		
69	DES					34		
70	INX					08		
71	INS					31		
72	LDX	CE	DE	EE	FE			
73	LDS	8E	9E	AE	BE			
74	STX		DF	EF	FF			
75	STS		9F	AF	BF			
76	TXS					35		
77	TSX					30		
78	BRA						20	
79	BCC						24	
80	BCS						25	
81	BEQ						27	
82	BGE						2C	
83	BGT						2E	
84	BHI						22	
85	BLE						2F	
86	BLS						23	
87	BLI						2D	
88	BMI						2B	
89	BNE						26	
90	BVC						28	
91	BVS						29	
92	BPL						2A	
93	BSR						8D	
94	JMP			6E	7E			
95	JSR			AD	BD			
96	NOP					01		
97	RTI					3B		
98	RTS					39		

Appendix

7 The Motorola 6800 Instruction Set[1]

MPU INSTRUCTION SET

The MC6800 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6).

MPU ADDRESSING MODES

The MC6800 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses this location when it fetches the immediate instruction for execution. These are two or three byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three byte instructions.

Indexed Addressing — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Implied Addressing — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two byte instructions.

[1]Courtesy of Motorola Semiconductor Products, Inc.

TABLE 2 – MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ABA	Add Accumulators
ADC	Add with Carry
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BCC	Branch If Carry Clear
BCS	Branch If Carry Set
BEQ	Branch If Equal to Zero
BGE	Branch If Greater or Equal Zero
BGT	Branch If Greater than Zero
BHI	Branch If Higher
BIT	Bit Test
BLE	Branch If Less or Equal
BLS	Branch If Lower or Same
BLT	Branch If Less than Zero
BMI	Branch If Minus
BNE	Branch If Not Equal to Zero
BPL	Branch If Plus
BRA	Branch Always
BSR	Branch to Subroutine
BVC	Branch If Overflow Clear
BVS	Branch If Overflow Set
CBA	Compare Accumulators
CLC	Clear Carry
CLI	Clear Interrupt Mask
CLR	Clear
CLV	Clear Overflow
CMP	Compare
COM	Complement
CPX	Compare Index Register
DAA	Decimal Adjust
DEC	Decrement
DES	Decrement Stack Pointer
DEX	Decrement Index Register
EOR	Exclusive OR
INC	Increment
INS	Increment Stack Pointer
INX	Increment Index Register
JMP	Jump
JSR	Jump to Subroutine
LDA	Load Accumulator
LDS	Load Stack Pointer
LDX	Load Index Register
LSR	Logical Shift Right
NEG	Negate
NOP	No Operation
ORA	Inclusive OR Accumulator
PSH	Push Data
PUL	Pull Data
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTS	Return from Subroutine
SBA	Subtract Accumulators
SBC	Subtract with Carry
SEC	Set Carry
SEI	Set Interrupt Mask
SEV	Set Overflow
STA	Store Accumulator
STS	Store Stack Register
STX	Store Index Register
SUB	Subtract
SWI	Software Interrupt
TAB	Transfer Accumulators
TAP	Transfer Accumulators to Condition Code Reg.
TBA	Transfer Accumulators
TPA	Transfer Condition Code Reg. to Accumulator
TST	Test
TSX	Transfer Stack Pointer to Index Register
TXS	Transfer Index Register to Stack Pointer
WAI	Wait for Interrupt

		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			IMPLIED				5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				M̄ → M	•	•	↕	↕	R	S
	COMA													43	2	1	Ā → A	•	•	↕	↕	R	S
	COMB													53	2	1	B̄ → B	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	(1)	(2)
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	(1)	(2)
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	(1)	(2)
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	(3)
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	•	•	↕	↕	(4)	•
	DECA													4A	2	1	A − 1 → A	•	•	↕	↕	(4)	•
	DECB													5A	2	1	B − 1 → B	•	•	↕	↕	(4)	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	(5)	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	(5)	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	(5)	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M (C ← b7 ← b0)	•	•	↕	↕	(6)	↕
	ROLA													49	2	1	A	•	•	↕	↕	(6)	↕
	ROLB													59	2	1	B	•	•	↕	↕	(6)	↕
Rotate Right	ROR							66	7	2	76	6	3				M (C → b7 → b0)	•	•	↕	↕	(6)	↕
	RORA													46	2	1	A	•	•	↕	↕	(6)	↕
	RORB													56	2	1	B	•	•	↕	↕	(6)	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M (C ← b7 ← b0 ← 0)	•	•	↕	↕	(6)	↕
	ASLA													48	2	1	A	•	•	↕	↕	(6)	↕
	ASLB													58	2	1	B	•	•	↕	↕	(6)	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M (b7 → b0 → C)	•	•	↕	↕	(6)	↕
	ASRA													47	2	1	A	•	•	↕	↕	(6)	↕
	ASRB													57	2	1	B	•	•	↕	↕	(6)	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M (0 → b7 → b0 → C)	•	•	R	↕	(6)	↕
	LSRA													44	2	1	A	•	•	R	↕	(6)	↕
	LSRB													54	2	1	B	•	•	R	↕	(6)	↕
Store Acmltr	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subtract Acmltrs	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A − 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B − 00	•	•	↕	↕	R	R
																		H	I	N	Z	V	C

LEGEND

OP Operation Code (Hexadecimal)
~ Number of MPU Cycles
\# Number of Program Bytes
\+ Arithmetic Plus
− Arithmetic Minus
• Boolean AND
M_{SP} Contents of memory location pointed to be Stack Pointer
\+ Boolean Inclusive OR
⊕ Boolean Exclusive OR
M̄ Complement of M
→ Transfer Into
0 Bit = Zero
00 Byte = Zero

Note – Accumulator addressing mode instructions are included in the column for IMPLIED addressing.

CONDITION CODE SYMBOLS

H Half-carry from bit 3
I Interrupt mask
N Negative (sign bit)
Z Zero (byte)
V Overflow, 2's complement
C Carry from bit 7
R Reset Always
S Set Always
↕ Test and set if true, cleared otherwise
• Not Affected

TABLE 4 – INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

POINTER OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	IMPLIED OP	IMPLIED ~	IMPLIED #	BOOLEAN/ARITHMETIC OPERATION	COND CODE REG 5	4	3	2	1	0
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				(XH) – M, (XL) – (M + 1)	↕	↕	↕	↕	↕	•
Decrement Index Reg	DEX													09	4	1	X – 1 → X	•	↕	•	•	•	•
Decrement Stack Ptr	DES													34	4	1	SP – 1 → SP	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	X + 1 → X	•	↕	•	•	•	•
Increment Stack Ptr	INS													31	4	1	SP + 1 → SP	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				M → XH, (M + 1) → XL	•	↕	↕	R	↕	•
Load Stack Ptr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				M → SPH, (M + 1) → SPL	•	↕	↕	R	↕	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				XH → M, XL → (M + 1)	•	↕	↕	R	↕	•
Store Stack Ptr	STS				9F	5	2	AF	7	2	BF	6	3				SPH → M, SPL → (M + 1)	•	↕	↕	R	↕	•
Index Reg → Stack Ptr	TXS													35	4	1	X – 1 → SP	•	•	•	•	•	•
Stack Ptr → Index Reg	TSX													30	4	1	SP + 1 → X	•	•	•	•	•	•

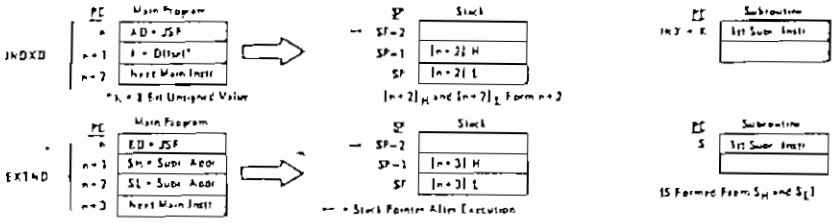
TABLE 5 – JUMP AND BRANCH INSTRUCTIONS

OPERATIONS	MNEMONIC	RELATIVE OP	RELATIVE ~	RELATIVE #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	IMPLIED OP	IMPLIED ~	IMPLIED #	BRANCH TEST	COND CODE REG 5	4	3	2	1	0
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										N ⊕ V = 0	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										Z + (N ⊕ V) = 0	•	•	•	•	•	•
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										Z + (N ⊕ V) = 1	•	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										N ⊕ V = 1	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										3B	10	1		———			⑫		———
Return From Subroutine	RTS										39	5	1		•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1	See Special Operations	•	•	⑪	•	•	•
Wait for Interrupt*	WAI										3E	9	1		•	•	⑪	•	•	•

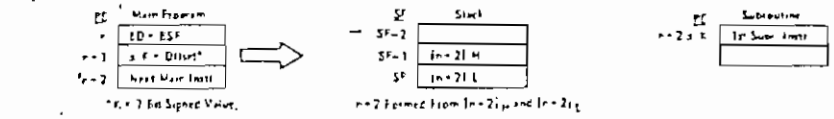
*WAI puts Address Bus, R/W, and Data Bus in the three state mode while VMA is held low.

SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE

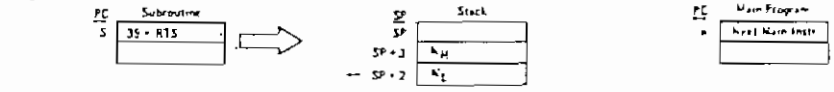


BSR, BRANCH TO SUBROUTINE*



JMP, JUMP

RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:

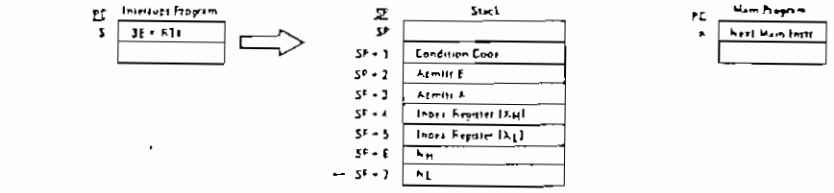


TABLE 6 — CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED OP	~	#	BOOLEAN OPERATION	COND CODE REG. H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	0 → C	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 → V	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	1 → C	•	•	•	•	S	
Set Interrupt Mask	SEI	0F	2	1	1 → I	•	S	•	•	•	•
Set Overflow	SEV	0E	2	1	1 → V	•	•	•	•	S	•
Accum A → CCR	TAP	06	2	1	A → CCR	(12)					
CCR → Accum A	TPA	07	2	1	CCR → A	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES (Bit set if test is true and cleared otherwise)

1 (Bit V) Test: Result = 10000000?
2 (Bit C) Test: Result = 00000000?
3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set)
4 (Bit V) Test: Operand = 10000000 prior to execution?
5 (Bit V) Test: Operand = 01111111 prior to execution?
6 (Bit V) Test: Set equal to result of N⊕C after shift has occurred
7 (Bit N) Test: Sign bit of most significant (MSP) byte = 1?
8 (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
9 (Bit N) Test: Result less than zero? (Bit 15 = 1)
10 (All) Load Condition Code Register from Stack. (See Special Operations)
11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
12 (ANI) Set according to the contents of Accumulator A.

TABLE 7 -- INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES
(Times in Machine Cycles)

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative
ABA		•	•	•	•	•	2	•
ADC	x	•	2	3	4	5	•	•
ADD	x	•	2	3	4	5	•	•
AND	x	•	2	3	4	5	•	•
ASL		2	•	•	6	7	•	•
ASR		2	•	•	6	7	•	•
BCC		•	•	•	•	•	•	4
BCS		•	•	•	•	•	•	4
BEA		•	•	•	•	•	•	4
BGE		•	•	•	•	•	•	4
BGT		•	•	•	•	•	•	4
BHI		•	•	•	•	•	•	4
BIT	x	•	2	3	4	5	•	•
BLE		•	•	•	•	•	•	4
BLS		•	•	•	•	•	•	4
BLT		•	•	•	•	•	•	4
BMI		•	•	•	•	•	•	4
BNE		•	•	•	•	•	•	4
BPL		•	•	•	•	•	•	4
BRA		•	•	•	•	•	•	4
BSR		•	•	•	•	•	•	8
BVC		•	•	•	•	•	•	4
BVS		•	•	•	•	•	•	4
CBA		•	•	•	•	•	2	•
CLC		•	•	•	•	•	2	•
CLI		•	•	•	•	•	2	•
CLR		2	•	•	6	7	•	•
CLV		•	•	•	•	•	2	•
CMP	x	•	2	3	4	5	•	•
COM		2	•	•	6	7	•	•
CPX		•	3	4	5	6	•	•
DAA		•	•	•	•	•	2	•
DEC		2	•	•	6	7	•	•
DES		•	•	•	•	•	4	•
DEX		•	•	•	•	•	4	•
EOR	x	•	2	3	4	5	•	•

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
INC		2	•	•	6	7	•
INS		•	•	•	•	•	4
INX		•	•	•	•	•	4
JMP		•	•	•	3	4	•
JSR		•	•	•	9	8	•
LDA	x	•	2	3	4	5	•
LDS		•	3	4	5	6	•
LDX		•	3	4	5	6	•
LSR		2	•	•	6	7	•
NEG		2	•	•	6	7	•
NOP		•	•	•	•	•	2
ORA	x	•	2	3	4	5	•
PSH		•	•	•	•	•	4
PUL		•	•	•	•	•	4
ROL		2	•	•	6	7	•
ROR		2	•	•	6	7	•
RTI		•	•	•	•	•	10
RTS		•	•	•	•	•	5
SBA		•	•	•	•	•	2
SBC	x	•	2	3	4	5	•
SEC		•	•	•	•	•	2
SEI		•	•	•	•	•	2
SEV		•	•	•	•	•	2
STA	x	•	•	4	5	6	•
STS		•	•	5	6	7	•
STX		•	•	5	6	7	•
SUB	x	•	2	3	4	5	•
SWI		•	•	•	•	•	12
TAB		•	•	•	•	•	2
TAP		•	•	•	•	•	2
TBA		•	•	•	•	•	2
TPA		•	•	•	•	•	2
TST		2	•	•	6	7	•
TSX		•	•	•	•	•	4
TSX		•	•	•	•	•	4
WAI		•	•	•	•	•	9

NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles

RRO

AT FILE 5?> NERRO

INICIALIZACION DEL ARCHIVO ? (S/N)

LECTURA DEL ARCHIVO NERRO ? (S/N)

SIGNIFICADO DEL CODIGO DE ERROR.....

ETIQUETA ERRONEA : UNA INSTRUCCION QUE NO PUEDE LLEVAR ETIQUETA ESTA ETIQUETADA.

ETIQUETA ILEGAL : UNA ETIQUETA USADA NO ESTA PERMITIDA POR LAS REGLAS DEL ASSEMBLER.

FALTA ETIQUETA : UNA INSTRUCCION QUE REQUIERE UNA ETIQUETA NO LA TIENE.

DEFINICION MULTIPLE : UNA ETIQUETA O SIMBOLO HA SIDO DEFINIDO MAS DE UNA VEZ.

FALTA 'NAM' : PRIMERA LINEA DEL PROGRAMA DEBE CONTENER LA PSEUDO OPERA - CION 'NAM'.

FALTA 'ORG' : SEGUNDA LINEA DEL PROGRAMA DEBE CONTENER LA PSEUDO OPERA - CION 'ORG'.

NO DIRECCIONAMIENTO : EL DIRECCIONAMIENTO NO CORRESPONDE A NINGUN TIPO PERMITIDO POR LA GRAMATICA DE ESTE ASSEMBLER.

SIMBOLO INDEFINIDO : UN SIMBOLO USADO NO HA SIDO DEFINIDO.

FORMATO ILEGAL : UN FORMATO INCORRECTO HA SIDO UTILIZADO.

EXPRESION INVALIDA : UNA EXPRESION HA SIDO FORMADA DE UNA MANERA ILE - GAL. PROBLEMAS COMUNES INCLUYEN : OPERACIONES IN - VALIDAS (ERRORES DE DIGITACION O SIMBOLOS INCO - RRECTOS), ARGUMENTOS OMITIDOS, ARGUMENTOS DE TI - PO O LONGITUD NO PERMITIDA, ETC.

FALTA OPERANDO : UNA INSTRUCCION QUE REQUIERE UN OPERANDO NO LO TIENE.

CODIGO DE OPERACION INDEFINIDO : UN CODIGO DE OPERACION USADO NO SE EN - CUENTRA EN EL SET DE INSTRUCCIONES. ESTE ERROR USUALMENTE SIGNIFICA QUE EL CODIGO DE OPERACION ESTA MAL DELETREA - DO, MAL UBICADO, U OMITIDO.

TABLA DE SIMBOLOS EXCEDIDA : LOS LIMITES FISICOS DEL ASSEMBLER NO SE EN- CUENTRAN DEFINIDOS COMO PARA PERMITIR EL NUMERO DE SIMBOLOS UTILIZADOS, EL VALOR DEL CONTADOR DEL PROGRAMA, ETC.

CARGA MANUAL DEL ARCHIVO NERRO ? (S/N).

B I B L I O G R A F I A

- Lance A. Leventhal (1978),- "6800 Assembly Language Programming",- Osborne & Asocciates.

- Alfred V. Aho, Jeffrey D. Ullman (1979), -"Principles of Compiler Design",- Addison - Wesley Series.

- David Gries (1975),-"Construcción de Compiladores",- Paraninfo.

- P.M. Leuvis II, D. J. Rosenkrantz, R.E. Stearns (1978),- "Compiler Design Theory",- Osborne & Associates.

- Lance A. Leventhal (1978),-"Introduction to Microprocessors: Software, Hardware, Programming",- Prentice Hall.

- Cytos DM-250 (1978),- Reference Manual.

- DNA FORTRAN IV (1978),- Reference Manual.